Implementing CAIA Delay-Gradient in Linux

Kenneth Klette Jonassen

May 2015

Submitted in partial fulfillment of the requirements for the degree of:
Master of Science in Informatics: programming and networks

May 2015

University of Oslo
Faculty of Mathematics and Natural Sciences
Department of Informatics
Oslo, Norway

Based on research carried out at:
Simula Research Laboratory
Fornebu, Norway

Abstract

We have implemented and evaluated an independent implementation of the CAIA Delay-Gradient congestion control in the Linux operating system. We made several adjustments or improvements to the design of CDG in our implementation. We have found sources of noise in the FreeBSD implementation of CDG. We identified areas of improvement to Linux' RTT measurements for congestion control. Our results indicate that our Linux implementation can compete effectively, and that it may operate more effectively than the FreeBSD implementation in terms of obtained throughput when it is not competing. Finally, we concluded that CDG is safe for use in the Internet.

Foreword

I am grateful for the support of my supervisors, Dr. Andreas Petlund, Prof. Pål Halvorsen, and Prof. Carsten Griwodz, for their encouragement and mentoring on this topical subject, and for their help with preparing this thesis. I am also thankful to Dr. David Hayes for taking the time to clarify and provide insight on parts of the CAIA Delay-Gradient mechanisms.

Yours sincerely,
Kenneth Klette Jonassen

Contents

Chapter 1

Introduction

A computer network provides a service that transports data from a sender to a receiver. Any such network, being a concrete and tangible entity, has a physical limitation to its maximum capacity for transporting data, and senders that exceed the network's limits will give rise to *congestion* in the network. Modest amounts of congestion can cause a performance degradation in terms of lost packets and undue delays, and more extreme amounts of congestion can cause a *congestion collapse*, where adding new data to the network can disrupt the preceding data from being delivered. It is mostly up to the sender and the receiver to avoid congestion in the network. Systems today employ *congestion control* to manage the amount of data put into the network by monitoring the amount of received data (Figure 1.1). We require that this mechanism helps us avoid the dire situation of a congestion collapse, but this is the bare minimum. Congestion controls would ideally do more in terms of avoiding loss and keeping delays low. In this thesis, we explore some of these possibilities through the implementation and evaluation of the CAIA Delay-Gradient congestion control in the Linux operating system.

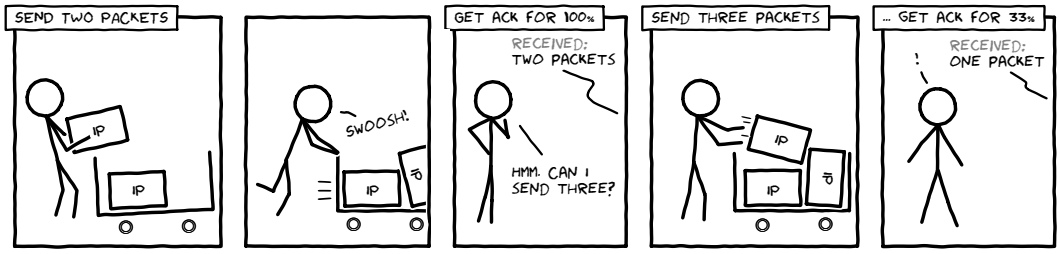


Figure 1.1: Illustration of a congestion control process, as seen from the sender. Through trial and error, the sender gradually oversteps, and eventually sends more than the network is able to carry.

1.1 Motivation

The Internet is perhaps the most ubiquitous and extensive computer network that is currently in use, and congestion control is a crucial component to maintain its stable and efficient operation [50]. However, the predominant congestion control mechanisms in use on the Internet today are far from perfect solutions to a very difficult problem. These rely on lost packets as feedback that they are sending too fast, and are thus inflicting a certain level of "gratuitous congestion" in order to effect that feedback. This is the approach taken by TCP's standard congestion control, but also prominent advancements such as CUBIC [30, 2]. Such *loss-based* congestion controls tend towards creating high delays and losses in the network that otherwise are potentially avoidable. However, Jacobson's argument for proposing loss to signal congestion, that "this signal is delivered automatically by all existing networks, without special modification", are all but historical today [42]. The argument was conceived while the Internet was very much in its infancy, and in hindsight, it might have been possible to upgrade the entire Internet at that time. But it would have been difficult to foretell the full implications of such design choices at that time, and how the Internet would grow to be as we know it today. Later efforts to deploy new signaling, such as Explicit Congestion Notification (ECN), has been hindered by compatibility issues with existing network equipment, and is typically unsupported by the network or disabled by default [50].

There is an alternative means of detecting congestion, without upgrades in the network, that has received attention from researchers since the late 1980s. We can observe that delays start to grow at the mere onset of congestion in the network, and by measuring this delay, we have a signal that can detect congestion before losses occur. Such *delay-based* congestion controls have the ability to send at speeds close to the network's limit, while potentially inducing less delay and no losses. Proposals include Jain's seminal delay gradient technique, the well-known TCP Vegas, and the more exotic TCP Veno [70, 63]. Several flavors of delay-based congestion control have been proposed, but they generally fall short of viable solutions [70, 34]. Changes in delay are not necessarily due to congestion, and in that sense, the delay signal can carry a certain amount of noise that is not correlated with congestion. Delay-based congestion controls also tend to compete poorly when there is bandwidth contention, readily giving up their fair share to others. This is either because they react much faster than loss-based flows, or because new delay-based flows fail to accurately detect their presence. We conclude that delay-based congestion control has potential to do more in terms of avoiding loss and keeping delays low, but that the established delay-based algorithms

have technical challenges that need to be addressed.

1.2 Problem

CAIA Delay-Gradient (CDG) is a flavor of delay-based congestion control that has seen interesting results in literature [34, 33]. It provides the benefits of an early congestion response using delay as a congestion signal, but more specifically, it employs novel mechanisms that potentially elude the issues inherent to earlier delay-based congestion controls. We are interested in delay-based congestion control because it has the potential to lessen delays and losses in networks such as the Internet, and more specifically, we are interested in CDG because it can solve or provide insight to the issues that have been inherent to delay-based congestion control. There is currently an implementation publicly available in recent FreeBSD distributions, but not in Linux. A linux implementation would make CDG available to a wider audience, and possibly encourage interest and future research on the topic.

In this thesis, our goal is to implement the CDG congestion control in the Linux kernel. Using the FreeBSD implementation as a reference, we explore differences that can affect their performance, make adjustments to take advantage of Linux' congestion control infrastructure, and provide improvements when applicable. This work applies directly to the Transmission Control Protocol (TCP), but can in part be transferable to other transport protocols such as RTP Media Congestion Avoidance Techniques (RMCAT), MultiPath TCP (MPTCP), Stream Control Transmission Protocol (SCTP), or Google's Quick UDP Internet Connections (QUIC).

1.3 Results and contributions

During our work to implement CDG in Linux, we encountered several challenges that led to results and contributions on related topics:

- We identified areas of improvement for Linux' RTT measurements, namely the absence of RTT measurements from SACK, and a special case that produced an erroneous RTT measurement.
- We identified sources of noise in FreeBSD's RTT measurements, while trying to explain a performance difference between Linux and FreeBSD implementations.
- We implemented CDG as a Linux module that takes advantage of Linux' congestion control infrastructure. We made several improvements

compared with the FreeBSD implementation, including a more accurate backoff, and a toggle that may improve CDG's utility for background or scavenger traffic.

- We identified areas of future work for CDG, and for congestion control in general.

Table 1.1 lists the patches that we submitted to the Linux kernel. An initial version of our CDG implementation was posted for review on the network development mailing list, and we received encouraging suggestions for improvements that need to be addressed before a final version can make its way into the Linux kernel.

Commit	Description	Kernel
3725a26	pkt_sched: fq: avoid hang when quantum 0	3.19
–	pkt_sched: fq: avoid artificial bursts for clocked flows	[1]
932eb76	tcp: use RTT from SACK for CC	3.19
3d0d26c	tcp: fix bogus RTT when retransmissions are ACKed	4.0
196da97	tcp: move struct tcp_sacktag_state to tcp_ack()	4.1
31231a8	tcp: improve RTT from SACK for CC	4.1
138998f	tcp: invoke pkts_acked hook on every ACK	4.1
	tcp: add CDG congestion control	(pending)

Table 1.1: Enhancements and fixes to the Linux kernel. Details in §C.

Outline

In Chapter 2, we briefly cover aspects of Internet congestion control with an emphasis on TCP.

[1] Collided with internal patch at Google ("net-sched-fq: special case low rate flows").

Chapter 2

Internet congestion control

This chapter briefly touches the background material required to present this thesis document as a self-contained work on aspects of TCP congestion control for use in the Internet. We assume readers to have background knowledge on par with an introductory course in computer networks, and emphasize breadth of material over depth of material.

2.1 The Internet Architecture

The Internet is a network of diverse interconnected networks that spans much of our planet. Users of the Internet – real persons and automated services alike – are situated at the edge or end of the Internet. Devices at the endpoints are known as hosts, and they use common protocols to communicate with each other over the network [12].

TCP/IP model	OSI model	Data Units	Protocols
Application Layer	7. Application Layer	Data, ...	HTTP, ...
	6. Presentation Layer	Data, ...	TLS, ...
	5. Session Layer	Data, ...	RPC, ...
Transport Layer	4. Transport Layer	Segment, ...	TCP, UDP, ...
Internet Layer	3. Network Layer	Packet, ...	IP, IPv6, ...
Link Layer	2. Data Link Layer	Frame	Ethernet, ...
	1. Physical Layer	Bit	CSMA, ...

Table 2.1: Conceptual models that guide protocol design and classification: Internet Protocol Suite (TCP/IP model) and OSI reference model [12, 67].

Each protocol is conceived as a layer depicted in Table 2.1. Except for the top and bottom, each protocol performs a certain function as a service for the layer above it, and utilizes the layer below it to do so. Internet hosts have no strict requirements such as size, speed or function, but communication with other hosts needs at least one protocol for each layer in the TCP/IP model [12]. Routers, switches and other devices between hosts will typically perform only functions of the Network Layer and below [7]. The most relevant layers for this thesis are the Network Layer and Transport Layer, where IP belongs to the former and TCP to the latter.

Network Layer – The Internet Protocol

The Internet Protocol (IP or IPv6) provides functions that are necessary to carry bits of data end-to-end, such as the addressing of source and destination over the Internet. These functions are *best-effort*, without guarantees for data integrity, reliability or sequencing [54]. Transmitted IP packets may be damaged, lost, duplicated or reordered, e.g., damaged due to bit errors at lower layers, lost due to network overload, duplicated due to loops or reordered due to IP route change.

Transport Layer – The Transmission Control Protocol

The Transmission Control Protocol (TCP) provides functions for reliable, in-sequence process-to-process communication on top of the network layer. The basic operations of TCP are:

- Establishment and termination of stateful process-to-process connections. Initiating processes are conventionally called *clients*, and processes listening for connections are called *servers*. Connections are multiplexed by means of source and destination port numbers. A pair of ports identifies segments sent from client to server, and swapping their order identifies responses in the converse direction.

- Providing full-duplex data transfer to the application layer as a sequential byte stream. A sender breaks the stream into discrete segments for transfer over the Network Layer, and a receiver reassembles the stream for in-sequence delivery to the Application Layer.

- Providing reliability by means of assigning sequence numbers to every byte in transit, acknowledging received sequences, and retransmitting unacknowledged sequences.

- Providing integrity by means of checksumming segment data.

- Providing flow control to avoid overloading the receiving process, and congestion control to avoid congestion collapse in the network.

Conclusion

This section summarized basic terminology of the Internet architecture and the services that IP and TCP provides. We have omitted in-depth descriptions or discussions, and instead refer readers that are unfamiliar with the subject of computer networks to excellent teaching material [70, 67].

The following section gives an account for the conception of TCP congestion control, briefly describes the rationale and operation of two fundamental congestion control algorithms in standard TCP, and introduces a concept of "fairness" that can be useful for evaluating the utility of congestion controls.

2.2 TCP congestion control

Timed and automatic retransmission was originally proposed as a TCP's sole solution to errors and congestion in the network [54]. This approach works in small networks, but was later observed to be inadequate for use in the Internet: when the collective demand of several senders filled multiple queues in the network, packets or their acknowledgements were observed to accumulate great amounts of queueing delay. Packets would get so excessively delayed that senders automatically began retransmission of packets that were not actually lost. This led to a deteriorating state where the network was fully utilized, but only a small fraction of its packets were useful, and the rest were spurious retransmissions. We know this as the *classic* case of congestion collapse, and use a more general definition [23]:

Definition 2.1. Congestion collapse occurs when an increase in the network load results in a decrease in the useful work done by the network.

The classic case was solved by the advent of Van Jacobson's congestion control efforts to TCP, but a second form of congestion collapse can potentially arise if enough packets are dropped in the network [23]. The resources expended when transferring a packet to the point of drop are wasted, and these might in turn have been resources that were necessary for other packets to complete their transfer, i.e., a packet can obstruct other packets without itself arriving at its final destination. Such a congestion collapse can only occur if there are two or more bottlenecks in the network [23]. Possible causes could be that a congestion control is faulty, or if senders overload the network without employing congestion control at all [2, 23].

Conservation of packets

A TCP sender tracks the number of transmitted and unacknowledged segments; the number of packets it has in flight is typically compared to a window when determining whether a TCP sender can transmit a new segment or not.[1] This restriction helps TCP avoid congestion collapse by enforcing the conservation of packets principle: a sender must not transmit more packets until it has evidence that original packets are no longer transiting the network [41].[2] The window *wnd* is governed by flow control and congestion control in unity:

$$wnd = \max(rwnd, cwnd), \tag{2.1}$$

where *rwnd* in the latest receive window announced by a TCP receiver for flow control, and *cwnd* is the congestion window. A TCP sender must employ two standardized algorithms to set *cwnd* and control the amount of data it injects into the network: slow start and congestion avoidance [2]. They are mutually exclusive – only one algorithm is used at any given time.

Slow start

The intention of slow start is to probe the maximum sustainable sending rate reasonably fast when the network environment is unknown. A sender should do so by doubling the congestion window *cwnd* for each round-trip time (RTT) that proceeds until packet loss is detected [2]. Once the sending rate is known, the sender must switch to congestion avoidance as to not unjustly inhibit other flows that are competing for network resources.

A sender should implement slow start as growing *cwnd* by the number of successfully and sequentially acknowledged segments. When packet loss is detected, it reduces *cwnd* by half and sets the slow start threshold *ssthresh* to this value. Since it takes one RTT from sending a packet until feedback can be received, reducing *cwnd* by half effectively restores its value prior to sending the lost packet. This value, *ssthresh*, is the last sending rate that is known to be sustainable. Slow start should be used to start a new connection, to restart transmission after a long idle period, or to restart transmission after a retransmission timeout (RTO) [2].

[1] Algorithms may extend or artificially constrain the window in some cases.

[2] Segments can be retransmitted without being lost when RTT estimates are inaccurate. Such *spurious retransmits* can break the enforcement of this principle. This problem is alleviated by reducing *cwnd* on retransmit. Forward RTO-Recovery (F-RTO) is an algorithm that restores *cwnd* and exits recovery when spurious retransmissions are detected.

Congestion avoidance

The intention of congestion avoidance is to maximize utilization of the network. Although slow start is initially used for a similar purpose, congestion avoidance is used to probe for available resources over time. This is useful when, for example, a flow ceases sending, making its share of resources available for competing flows.

Using congestion avoidance, the sender extends its congestion window *cwnd* less aggressively than slow start: the rough idea is lengthening *cwnd* by one segment for each RTT that proceeds without detectable congestion. From a newly injected segment is transmitted, it takes exactly one such RTT to receive its acknowledgement. Since this is feedback supporting that the current sending rate is sustainable, the sender can progressively repeat the process in good faith.

2.2.1 Fairness

When multiple flows are using the network at the same time, they are possibly competing with each other for the available bandwidth. Their ability to compete for bandwidth is affected by several factors, including how "aggressive" they are when competing. For example, traditional delay-based congestion controls may easily get "outmuscled" in competition with loss-based congestion controls as described in §1.1.

We can use a concept of fairness to evaluate the allocation of network resources, where the level of fairness is expressed using a decimal number ranging from 0 to 1. A value close to 0.00 indicates no fairness, while a value close to 1.00 indicates maximum fairness. Different forms of fairness can be systematically estimated and compared using measures that give such a value.

Informally, *flow fairness* describes whether some flows consume more or less resources than other flows. *Jain's fairness index* [24] measures the parity of rate allocations between flows (flow rate fairness):

$$(\sum_{i=1}^{n} x_i)^2/(n \sum_{i=1}^{n} x_i^2), \tag{2.2}$$

where n is the number of competing flows, and x_i is the throughput of the ith flow. If j flows are not receiving any allocations (starving), the fairness index is $(n - j)/n$, and allocations are less than maximally fair. This measure is suited when there is a contention for resources in the network, and all the accounted flows are in equal need of resources, i.e., the unfair flows are the ones too greedy when all the flows are needy.

Discussion

We gave an account for the conception of TCP congestion control, and briefly described the rationale and operation of slow start and congestion avoidance algorithms in standard TCP [2]. A sender uses slow start when $cwnd < ssthresh$, or congestion avoidance when $cwnd > ssthresh$. Either algorithm can be chosen when $cwnd = ssthresh$; the default congestion control in Linux selects slow start for this case. New congestion controls may change parts of slow start and congestion avoidance algorithms within the confinements of standardized behavior in RFC 5681. Usage of the keyword *should* in the standard is not to be confused with a requirement, since it only indicates a well-founded recommendation that can be overruled when appropriate [13]. Literature describes the standard slow start and congestion avoidance algorithms as strategies of Multiple Increase Multiplicative Decrease (MIMD), and Additive Increase Multiplicative Decrease (AIMD), respectfully [70]. This naming corresponds to their adjustment of *cwnd* over one RTT, where standard TCP (AIMD) uses Additive Increase (AI) in absence of congestion indications, and Multiplicative Decrease (MD) when packet loss is detected.

We have omitted descriptions of the congestion control algorithms Fast Retransmit and Fast Recovery [2]. The congestion control known as *Reno* implements all four aforementioned algorithms as specified in RFC 5681, and its successor *NewReno* additionally implements the Fast Recovery modification in RFC 6582. We will not discuss recovery mechanisms in this thesis, and refer interested readers to RFC documents [2, 40] for their complete description.

A TCP congestion control traditionally adjusts *cwnd* to regulate its sending rate, but its actual resource consumption in the network is also affected by variables such as the RTT. We can instead use throughput as a proxy for the resources that a flow consumes in the network, and this is what Jain's fairness index traditionally uses to measure flow rate fairness [24]. There is a distinction between the fairness measure that we presented here, and those used in for example the social sciences [15]. Flow fairness should not be confused with fairness amongst entities such as users or applications. Any entity can create an arbitrary large number of flows that "fairly" receive a proportionally large share of resources. Flow fairness does not accurately portray the conventional meaning of fair resource allocation, but we consider it useful for our experimental assessment of congestion controls.

The way that a congestion control interprets congestion signals to regulate its sending rate is tightly coupled to the mechanics of queueing and bottlenecks. In the next section, we give an introduction to some of the most important concepts of bottlenecks in the Internet.

2.3 Bottlenecks

Bottlenecks appear anywhere in the Internet – routers, switches and end systems alike – dropping or queueing packets where they occur. They are the result of a finite link capacity, processing limitation, or administrative QoS policy. This section touches on the rationale for queueing at bottlenecks, how these queues form during periods of congestion, and how different types of queueing behavior can affect transiting traffic through a bottleneck.

2.3.1 Queueing

A property inherent to packet-switching is statistical multiplexing. It allows several users to share a given transmission channel, and each user is given equal opportunity to grab the entire channel capacity. This can increase its utility as opposed to each user having a smaller, dedicated channel, but only one user can use a channel at any given instant. Statistical multiplexing thus also creates a breeding ground for resource contention, where several users compete for simultaneous access to the channel. A queue is a resource sharing mechanism that allows such conflicting demands to be served in some order [45].

Definition 2.2. *Congestion* occurs at bottlenecks in packet-switched networks when the instantaneous demand exceeds capacity, i.e., when two or more packets compete for the available capacity. *Congestion episodes* are periods of sustained congestion at bottlenecks, i.e., while queues hold more than one packet.

Congested bottlenecks either drop or queue packets that arrive. When reliable data transports are used, packet drops at a bottleneck lead to additional network resources being spent on retransmissions between the sender and the bottleneck. The network resources expended when transferring packets to the bottleneck are only preserved by queueing packets, and conversely dropping packets wastes these network resources. The use of queues can thus be shown to improve network efficiency.

2.3.2 Bufferbloat

Queues and buffers are conceptually different: a queue can form as packets organized in a buffer, but the buffer itself is only the space to hold packets [8]. Gettys et al. depict a scenario where increasingly larger buffers are being deployed in the Internet with little thought or testing as to how queues in such buffers should be managed [27]. Although the use of queues can improve

a network's efficiency, allowing queues to grow past a certain point has conversely shown to decrease a network's effectiveness. A TCP can continue to increase its sending rate until packet loss occurs, i.e., it requires queues to fill and drop packets in order to get feedback [48, 58]. In the absence of proper queue management, oversized buffers and TCP can thus lead to long queues and undue queueing delays that slows TCP's feedback loop [3, 71]. This interaction informally describes the phenomenon known as bufferbloat.

> "The existence of excessively large buffers inside the network is characterized as *bufferbloat*. These buffers frequently fill, and defeat the fundamental congestion avoidance algorithms of TCP." [27]

There are different incentives to keep queues short and limit bufferbloat. Empirical evidence supports that the use of large buffers relative to n induces synchronization in TCP flows that impedes efficient queueing [71, 56]. Synchronization occurs when the control loop of two or more flows are in phase, meaning that these flows react to congestion and reduce *cwnd* simultaneously.

Appenzeller et al. [3] used ns-2 simulations to show that the average flow completion time[3] for competing TCP flows is shorter with a reduced buffer size, and that this buffer still achieves full link utilization.[4]

Applications such as Voice over IP (VoIP) have a strict timeliness requirement due to their interactive nature [52, 66]. They value low delay, and packets are mostly useful when delivered expeditiously. In such cases, packet drops can even be preferable to a long queueing delay as to minimize clogging the bottleneck for future packets. This only applies when using a transport protocol that, unlike TCP, avoids head-of-line blocking, i.e., TCP's promise of in-order delivery to the application that requires lost segments to retransmit before data in following segments can be delivered.

The complexity and cost of router architectures limit the buffer sizes that can be feasibly produced [20]. Link speeds have grown faster than memory access speeds, requiring significant optimizations and expensive trade-offs in the architectural design of proportionally bigger and faster buffers for these link speeds [20, 10]. This precedent suggests that a reduced buffer capacity can translate to savings in the design and production costs of routers.

Optical Packet Switching (OPS) is a developing field of technology that process packets in the optical domain. It has promising applications, but the engineering challenge of buffering in OPS increases with the buffer capacity. Big buffers may be prohibitively difficult to produce for OPS [10, 26].

[3] The *flow completion time* is the time from when a flow's first packet is sent until the last packet reaches the destination.

[4] The simulation compared BDP and Appenzeller's rule, both described in §2.3.3.

Bufferbloat is an unfavorable phenomenon in the Internet, and we are motivated to limit bufferbloat for improvement in any of the aforementioned areas. Approaches to reduce bufferbloat include proper queue management in the network, and changes to congestion control at end hosts. Our interest is chiefly on the latter, but its operation is also closely intertwined with the former.

2.3.3 Queue management

Queue management informally describes any algorithm that manages the length of packet queues [11]. *Tail drop* specifically describes a queue management algorithm that drops arriving packets when their destined queue is full [6, 11]. There are two major performance considerations associated with tail drop [11]:

- Tail drop can adversely affect the fairness of competing TCP flows because of a *lock-out* phenomenon: tail drop favors flows already inside the queue, making it hard for flows at the tail to grab their fair share of queue space.
- Tail drop inhibits further growth of the queue only when it is full. The algorithm does not effectively manage queue length before reaching this state. Oversized queues using tail drop thus contribute to bufferbloat.

Because the gravity of lock-out and bufferbloat is proportional to queue length it can be beneficial to limit queueing below maximum buffer capacity [27, 11]. Commercial routers support setting the queueing limit used for tail drop as a configurable threshold parameter [66]. Guidelines for configuration depend on the link's capacity *bandwidth*, the number of active flows n, and the average flow round-trip time *delay*.

Nichols and Jacobson [49] make the distinction between good and bad queues: *good queues* convert bursty packet arrivals to smooth, steady departures while dissipating in about one RTT while *bad queues* persist for several RTTs. The *rule-of-thumb* or *bandwidth-delay product* (BDP) is widely used for sizing router buffers [3, 19, 49, 26]. The rationale is that a *single* TCP flow needs to have at least that much data in flight per round-trip to achieve full throughput, i.e., so that the bottleneck queue does not underflow when a TCP sender halves its congestion window after loss. By its very definition, the rule-of-thumb is a good queue for single TCP flows but can be inexpedient for large bandwidth-delay products.

For more than one flow, Appenzeller et al. propose that n active and *unsynchronized* TCP flows only require a queue length of $bandwidth \times delay/\sqrt{n}$

[3, 26]. It has been shown experimentally that TCP synchronization reduces link utilization when comparing this rule to BDP for low values of n [69, 3, 26]. Simulation analysis suggest that n should be 500 or greater for TCP flows to be sufficiently unsynchronized with this rule[5] [3].

Analysis and experiments indicate that even a tiny buffer of 20–50 packets can perform adequately in a core network given certain assumptions [26]. Such tiny buffers can be feasible for switching in the optical domain where a certain loss in transmission efficiency can be well tolerated in exchange for greater improvements in transmission capacity.

Active Queue Management

Active Queue Management (AQM) [11] is a class of technologies that proactively slow or throttle senders to manage queues in the network. They typically use congestion signaling such as drop or ECN marking to prevent senders from filling a queue.

Random Early Drop (RED) [70] is an early incarnation of AQM that probabilistically drops or marks arriving packets as a function of the average queue length, i.e., the probability gradually rises as queue lengths grow. Since packets are chosen at random, flows can be signaled of congestion at different points in time. This can help reduce lock-out and alleviate the synchronization of TCP flows seen when tail drop queues fill, reducing phase effects and improving the utility of Appenzeller's rule for buffer sizing.

Controlled Delay (CoDel) uses the minimum time that packets are queued over some interval to determine its drop or marking strategy [49]. This provides a way for load shedding based on long-term queueing delay and makes it easier than RED to allow for the intermittent bursts that characterize good queues.

Discussion

It is challenging to specify good parameters for n and rtt at bottlenecks in the Internet. Use of worst-case parameters is not an ideal solution as it adds to the bufferbloat problem [49, 27, 70]. And conversely, the use of undersized buffers leads to suboptimal link utilization [3, 71, 70]. The research on buffer sizing for TCP has been focused on traditional loss-based congestion control algorithms. Congestion control using delay or ECN signaling can potentially decrease phase effects, lending support to an hypothesis that

[5] Appenzeller et al. remark that some out-of-phase synchronization was visible in simulations up to $n < 1000$ but that buffer requirements are similar enough to disregard at this point.

such congestion control algorithms could work with Appenzeller's rule for $n < 500$, but research is needed to draw preliminary conclusions. Based on existing research on loss-based congestion control, specifying a queue length for tail drop necessitate a trade-off between throughput and delay, or a priori knowledge of the parameters such as when experimenting in controlled environments.

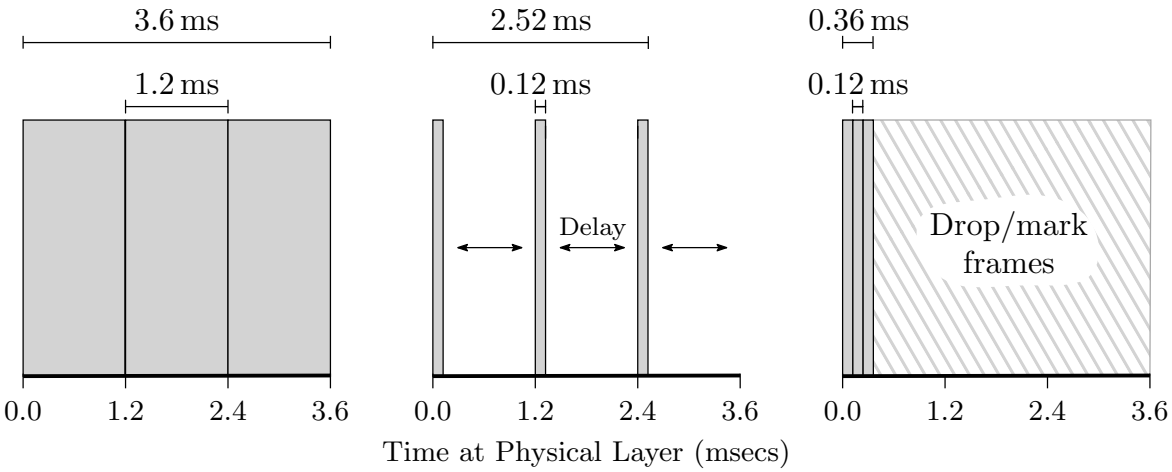
Since AQM can manage queues without tail drop, it alleviates the problem of configuring a tail drop threshold. Although AQM is recommended for deployment in the Internet, setting it up can be complicated [11, 9]. Jacobson has pointed out that queue length is an inaccurate proxy for load in network controls, suggesting that RED's load shedding is inaccurate [43]. Another known issue with RED is that its parameters need skillful tuning to achieve good performance. One of CoDel's selling points is that its default parameters can work in many environments, effectively proposing a plug and play solution to AQM. In any case, deployment of an AQM mechanism may be impeded by lack of support in switches and routers [66].

A general observation of queue management mechanisms are that they affect the performance of congestion controls at end hosts. For incremental deployability in existing networks, this could imply tuning queue management mechanisms to existing congestion controls. The inverse case of tuning a congestion control to a queue management mechanism has shown merit in private networks, e.g., DCTCP's marking threshold [1], or CDG's loss tolerance heuristic [4]. However, assuming the use of a particular queue management mechanism may not be applicable in diverse environments such as the Internet; for example, we cannot assume that the cause of congestion-related loss is tail drop due to AQM deployments.

2.3.4 Bandwidth

A link's bandwidth characterizes the rate it can transfer data. Bandwidth is bounded by capacity, but can also be limited below that capacity. This enables Internet Service Providers to roll out fiber-optic links with excessively high capacity to customers, accommodating future growth in bandwidth requirements without costly equipment upgrades. It also enables constraining certain traffic classes in order to enhance service quality for other traffic, e.g., limiting bandwidth consumed by data backups so that other traffic is not adversely affected.

Light waves and other signals can not be "slowed down" at the physical layer, but propagate at a fixed speed. Ethernet and other link layer protocols may negotiate different link speeds, but this can be insufficient or undesirable for limiting bandwidth. Shaping and policing are Quality of Service (QoS)



(a) No metering
10 mbps

(b) Shaping
100 → 10 mbps

(c) Policing
100 → 10 mbps

Figure 2.1: Delays for no metering, shaping, and policing. Bandwidth is 10 Mbps. Link capacity is 10 Mbps for no metering, and 100 Mbps for shaping and policing. Bars show transfer time for 1×1500 bytes and 3×1500 bytes.

mechanisms that provide a means to make decisions about traffic exceeding a certain rate [66, 6]. QoS has many uses – this section focuses on three:

No metering does not delay or drop packets before transmission. Packets are transmitted at full link capacity.

Shaping buffers packets to ensure smooth packet departure at a certain rate. Part of the traffic's burstiness is removed since shaping *delays* traffic to conform with a certain rate. Shaping closely approximates the behavior of a link with lower capacity, giving packets similar delay characteristics.

Policing *drops* or *marks* packets exceeding a certain rate. Policing does not increase packet delay and can clear a standing queue faster than shaping or links with lower capacity, giving policing better delay characteristics than shaping. Policing can also put more strain on the network compared to shaping as the sheer force of arriving bursts passes through without damping. Policing may interact poorly with TCP's self-clocking property and cause bigger bursts of dropped packets compared to shaping.

Figure 2.1 shows different delays induced by a congested bottleneck. The delays depend on link capacity, and whether congestion occurs due to limited link capacity or use of shaping or policing to enforce a sub-capacity bitrate.

Both shaping and policing depend on a metering mechanism to provide information about traffic characteristics. Metering may allow commencing traffic an initial burst that is transmitted at full link capacity, but this does not affect steady-state behavior described above. Metering can be implemented using token bucket or leaky bucket algorithms [16, 66]. These conceptual algorithms describe relevant details to this section, e.g., the rate of token replenishment, but the effects of their parameters are not evaluated in this thesis. Interested readers can find them in textbooks on computer networks [67, 70, 66].

2.4 Packet delay model

Packet delay is the total time taken to transport a packet from its sender to its receiver. It can be modeled as the sum of several distinct delays that a packet experiences [66]. A model helps us identify sources of delay, describe them, and estimate their contribution when we have knowledge about the network. Its parts are:

Serialization delay: time taken to place the entire frame onto the physical medium, given by *frame size/capacity*, e.g., serializing a 1514 byte Ethernet frame at 10 Mbps takes $1514 \times 8/(10 \times 10^3) = 1.2112$ ms.

Propagation delay: time taken for a signal to propagate through the physical medium, given by *distance/propagation time*. The propagation time varies based on physical characteristics, e.g., commercial fiber optic cables have a propagation time of roughly 2/3 to 3/4 the speed of light.[6]

Processing delay: time spent by all network elements processing the packet. Includes time taken to encapsulate the packet with headers.

Queueing delay: time spent in congested queues waiting for other packets to be sent.

Shaping delay: time spent waiting in front of the queue due to intentional head-of-line delay induced by traffic shaping. Literature varies between distinguishing it from queueing delay or considering both as one of the same [70, 66]. We consider shaping delay separately in this thesis since the experiments in §4 use shaping delay to simulate propagation delay.

[6] Propagation time through a fiber optic cable is given by *speed of light/refaction index*. Refraction index varies slightly between different makes of cables.

Serialization delay and propagation delay are typically fixed for a given capacity and frame size, regardless of using aforementioned QoS mechanisms such as shaping or policing; they give us a lower bound for the attainable packet delay under optimum network conditions. Queueing delay and shaping delay are interesting for congestion control since they can tell us something about congestion in the network. However, the individual parts can not be measured directly at end systems – only their sum. It can be difficult for end systems to distinguish processing delay from delays related to congestion since both are variable delays.

2.5 Congestion signals

A TCP uses receiver acknowledgements to learn about congestion in the network. We note that this can be seen as a signal processing problem [44]. In signal processing, a *signal* is a description of how one parameter depends on another parameter [64], e.g., packet loss is an inferred signal that describes if a given packet is lost or not [70].

This section is an overview of congestion signals that are either deployed or feasible for deployment in the Internet; it contains only those signals that a TCP can feasibly make use of, and omits noteworthy approaches that require intrusive upgrades in the network; e.g. eXplicit Congestion Protocol (XCP) proposes to modify routers so that senders are explicitly informed of which rate they may send [67].

Packet loss and delay do not explicitly inform of congestion per se, but these signals can be perceived to infer congestion, and are thereby featured as congestion signals implicitly.

2.5.1 Packet loss

The network can signal congestion to end hosts using packet loss, typically by means of tail drop or AQM action. The historic assumption by Van Jacobson is that packet loss uncorrelated with congestion is improbable, and thus packet loss must be interpreted as a signal of congestion [42, 41]. This assumption is challenged by physical media that are susceptible to noise or signal degradation, i.e., causing seemingly random corruption or loss at the physical layer [50]. Corruption effectively results in packet loss unless an optional redundancy coding is able to counteract it. Notable examples include:

- Telephone lines are prone to corrupt data. The signals that they transfer are subjected to crosstalk in multi-pair cables, and reflections at splice

points. They are extensively used in telephone networks for delivering Internet access via Digital Subscriber Line (DSL) technology.

- Mobile and wireless networks inflict corruption and loss, e.g., due to radio interference or base station handovers.

Reacting to packet loss is desirable when it relieves congestion in the network, but it also obstructs performance in networks where packet loss is frequent and only intermittently correlated with congestion. In §2.1, we described how standard TCP congestion control reacts to packet loss by effectively reducing its sending rate. Ignoring packet loss altogether is inadvisable; it is the only congestion signal that is invariably deployed in the Internet [70]. However, a congestion control that distinguishes loss due to congestion from loss due to other causes opens for conditioning this reaction [34].

2.5.2 Packet delay

Congestion correlates with delays induced by queueing or shaping, as described in §2.4. Measurements of packet delay provide a means to observe these delays indirectly. Delay-based senders can thus use trends in packet delay for reacting timely at the onset of queue buildup, rather than the incipient packet loss when queues fill up.

Packet delay can best be understood as a supplementary congestion signal. It requires assuming that congestion in the network can be observed through changes in packet delay, but this assumption is challenged:

- Events unrelated to congestion can change packet delay, such as rerouting in the network or link-layer ARQ [70].

- Packet delay is unaffected by some bottlenecks. Policed bottlenecks can drop packets that exceed a certain rate as described in §2.3.4. Queueing can still occur with policing, but may be in negligible amounts that are not reliably measured through delay.

- Packet delay has low correlation with congestion in highly multiplexed environments [55, 47]. Aliasing distortions in the delay signal grows with the level of multiplexing. It is argued that this is not an obstacle for congestion control since the aggregate behavior of delay-based senders are still adequate [47], but it illuminates a limitation.

Two types of packet delay measurements are known to be viable at end hosts:

Round-trip time (RTT): the time between sending a packet and receiving its acknowledgement as measured by the sender.

One-way delay (OWD): the packet delay as measured by a sender and receiver in collaboration.

RTT measurements can be performed by any standards-compliant TCP [44]. RFC 5681 recommends that TCP receivers combine acknowledgements to reduce traffic [2]. This is achieved by delaying the acknowledgement of received segments in anticipation of more arriving within a short timeframe. Such delayed acknowledgements can introduce systematic bias to sender-side RTT measurements and reduce the frequency of RTT sampling. This is especially an issue for low rate flows that only send solo segments at a time, i.e., the receiver delays acknowledgement until it hits an upper bound on allowed delay. However, TCP receivers are also recommended to send an immediate ACK for at least every second segment received [12, 2]; the impact of employing delayed acknowledgements as recommended could thus be expected to decrease when sending rates increase. Receivers that disregard the latter recommendation, i.e., delaying ACKs for more than two segments at a time, are said to employ Stretch ACKs [51].

Linux implements RTT measurements using local timing. A timestamp for transmitted segments is locally recorded in a linked list queue shared with retransmission data. RTT measurements are produced when data in the retransmission queue is acknowledged and removed [68].

An advantage of OWD in comparison to RTT is to avoid conflating delay in the forward and reverse path. This makes it a cleaner signal for detecting congestion as one path's delay fluctuations can not effect the other path. An implementation of OWD measurement requires support from both sender and receiver since calculation depends on knowing both departure and arrival timestamps of the packet in question.

Delay threshold

Delay threshold algorithms [34] base congestion avoidance decisions on whether a recent delay measurement is above or below a certain threshold. One approach is to estimate the path-specific minimum RTT, called the *base RTT*, as the absolute minimum of all measured RTTs. Estimations of queueing delay on the path are made by comparing recent RTT measurements with the base RTT.

TCP Vegas calculates the expected and actual throughput rate using the current congestion window *cwnd*, the absolute minimum *base_rtt* and the period minimum *min_rtt* [14]:

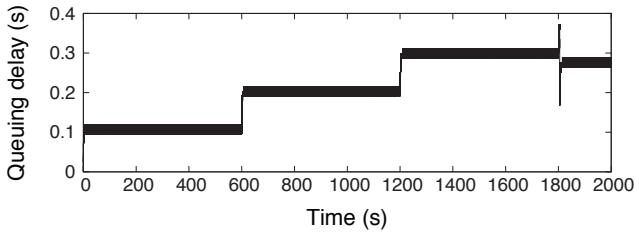


Figure 2.2: Base delay as estimated by LEDBAT, periodically increasing every 10 minutes [60]. Copyright © 2013 IEEE (Reused with permission).

$$\begin{aligned} expected &= cwnd/base\_rtt \\ actual &= cwnd/min\_rtt \end{aligned}, \quad \text{all variables} > 0. \tag{2.3}$$

The difference between these rates ϵ is compared with threshold parameters α and β once per elapsed RTT period to a make a decision between increase or decrease of the congestion window:

$$cwnd = \begin{cases} cwnd + 1 & \epsilon < \alpha \\ cwnd - 1 & \epsilon > \beta \\ cwnd & \text{otherwise} \end{cases}, \quad \epsilon = actual - expected. \tag{2.4}$$

If measurements are between thresholds, the congestion window is unchanged. The Vegas authors' suggestion is that $\beta - \alpha = 2MSS$ [14] and default values are $\alpha = 2MSS$ and $\beta = 4MSS$ in the Linux implementation [39]. Vegas never forgets the base RTT as currently implemented [36, 39]. This makes it incapable of adapting to increases in delay that are unrelated to congestion, e.g., an increase in propagation delay after rerouting.[7]

LEDBAT [63] tries to address this shortcoming of Vegas by keeping a sliding window of base delay (windowed minimum) that helps it forget over time. Simulations have shown an undesirable shortcoming of this approach when a single sender steadily sends enough data to cause a standing queue [60]. The sender will subsequently incorporate its self-induced queueing delay in the base delay estimate, creating a cycle that pushes new thresholds slightly upwards and allows it to create even larger queues. Figure 2.2 illustrates this cycle for a single flow with a window history going back 10 minutes.

[7] The TCP Vegas paper also proposes a modified slow start that increases *cwnd* at half the rate of Reno (every second RTT) and heuristically determines the initial *ssthresh* based on the spacing of acknowledgements. This is not currently implemented in Linux or FreeBSD.

The issue of estimating base delay also extends beyond self-induced queueing; newcomers to a shared bottleneck can not accurately establish a base delay if existing flows keep a standing queue. When Vegas flows compete, this coerces new flows to push queueing delays upward, and puts old flows at a competitive disadvantage because of their lower base delay estimate.

2.5.3 Explicit Congestion Notification

Definition 2.3. *Explicit Congestion Notification* (ECN) is a feedback mechanism where the network can signal hosts of incipient or occurring congestion without dropping packets or inducing packet delay.

Using explicit congestion signaling was discussed around the time Van Jacobson's classic congestion avoidance scheme was proposed [42]. An early approach that was suitable for IP and TCP is the ICMP Source Quench message [53]. The idea was that congested gateways would send explicit messages to the sender when dropping packets, as to irrefutably indicate congestion and expedite retransmissions [53]. An advantage compared to ECN is that no receiver cooperation was necessary, but the messages themselves contributed to congestion and were subject to spoofing attacks. Source quench was discouraged from deployment due to efficiency and security reasons [7]. Jain suggested a binary feedback scheme based on earlier work that addressed these concerns [59, 42]. ECN was later proposed as an experimental extension to IP [57] and source quench was deprecated [28] following the ratification of ECN [58].

ECN occupies two bits in IPv4 and IPv6 headers, refer to Table 2.2. Network elements indicate a congested destination by setting CE bits; this step is generally performed by an AQM algorithm. Receivers must relay this CE marking back to the sender. ECN signaling should only be used when supported by the transport protocol at both endpoints. Senders indicate support for ECN by setting either ECT(0) or ECT(1). Network elements are otherwise agnostic to how a receiver's transport protocol relays this congestion information back to the sender.

Any transport protocol may utilize ECN signaling, but the ratification of ECN also includes a specification of how a TCP should support ECN. *Classic ECN* denotes TCP's behavior as described in RFC3168 [58]. A TCP receiver relays one or more CE markings to the sender using TCP acknowledgements with the ECN Echo flag (ECE). It continues to send this flag for all subsequent acknowledgements until receiving a packet with the Congestion Window Reduced flag (CWR). This design only informs a sender that some packets were marked, not the exact count of packets. It serves to

Codepoint	Usage
00	Non-ECT: Sender does not support ECN
01	ECT(1): ECN Capable Transport 1
10	ECT(0): ECN Capable Transport 0
11	CE: Congestion Encountered

Table 2.2: ECN field in IP headers [58, p. 6]

reliably elicit the same congestion control response as packet loss, except no packets need be lost or retransmitted to do so.

A sender is free to choose ECT(0) or ECT(1) to indicate support for ECN, and may use this distinction for information. It has been proposed [65] that TCP receivers copy the sender's ECT bit so that senders can perform a compliance test. This test identifies rogue receivers that do not relay CE markings, since the CE marking overwrites ECT and makes receivers unaware of the sender's actual ECT nonce. Rogue receivers otherwise gain an unfair advantage as complying flows would get a penalty in the same situation. Another proposal is for the network to use more bits indicating the level of congestion, providing a finer feedback granularity.

The distinction of Classic ECN has a purpose for future TCP congestion control algorithms. The novelty of DataCenter TCP (DCTCP) is a congestion response based on fine-grained ECN feedback [1]. DCTCP modifies receivers to always relay the exact CE marking of incoming packets, albeit at the cost of ignoring the reliability function that Classic ECN employs. This makes a DCTCP sender vulnerable to loss of ACKs from the receiver, which it currently has no mechanism to detect or cope with [18]. Nonetheless, DCTCP has shown an attractive performance in private networks that can beat standard TCP in certain scenarios [1]. Algorithms that provide equivalent or better receiver feedback, and in an approach that is reliable, are currently an active research topic [18, 17, 50].

2.6 CAIA Delay-Gradient

CAIA Delay-Gradient (CDG) is a TCP congestion control originating from Swinburne University's Centre for Advanced Internet Architectures (CAIA). Its development was part of CAIA's efforts to implement a new congestion control framework in FreeBSD under the NewTCP project [4]. It has inspiration from Jain's CARD algorithm [34], the probabilistic backoff mechanism in

Hamilton Delay [4], and it borrows coexistence heuristics from CAIA-Hamilton Delay [34].

At time of writing, CDG is for experimental use and has not been through an RFC process as is recommended for new congestion controls [25]. This section describes CDG's design based on CAIA's paper and FreeBSD implementation [37, 34]. Default parameters mentioned are those used in CAIA's paper and CDG's FreeBSD implementation.

CDG extends standard TCP congestion control described in §2.1. It changes the TCP sender to:

1. Estimate the gradients of minimum and maximum delay using filtered packet delay measurements.

2. Use delay gradients to back off with an average probability that is independent of the RTT.

3. Improve performance in environments with seemingly random packet loss that is not caused by congestion.

4. Coexist with flows that use loss-based congestion control, i.e., flows that are unresponsive to the delay signal.

These changes work with unmodified TCP receivers, which makes CDG and non-CDG endpoints interoperable.

2.6.1 Delay gradients

A *delay gradient* estimates local queueing trends by observing change in packet delay measurements. These trends alone can be sufficient to detect congestion, thus eluding the base delay issues described in §2.5.2. Its independence of base delay makes it resilient in face of changes to the propagation delay, and gives it robustness against pre-existing congestion that bias base delay estimates. These issues are obstacles to deployment that have characterized other delay-based congestion controls.

CDG uses two gradients of delay to detect congestion in the network. Gradients g_{min} and g_{max} are obtained by filtering the minimum and maximum packet delay measured over two successive round-trip times:

$$g_{min}(n) = \min M_n - \min M_{n-1}, \quad g_{max}(n) = \max M_n - \max M_{n-1},$$

where M_n is the set of packet delay measurements for RTT interval $n \geq 2$. Filtering serves a functional purpose to obtain these distinctive gradients, but

can also reduce the effects that measurement noise and transient delays have on the signal [64].

When operating in congestion avoidance, the gradients are smoothed using a recursive moving average [64, 34]:

$$\hat{g}_n = \hat{g}_{n-1} + \frac{g_n - g_{n-w}}{w}, \quad g \in \{g_{min}, g_{max}\}, \quad w \geq 1, \tag{2.5}$$

where w is the window width (8 by default). Smoothing can reduce noise in the RTT signal, and helps with loss tolerance and coexistence heuristics [34].[8] When operating in slow start, gradients are used directly as input to CDG's backoff mechanism to ensure the most timely response to self-induced congestion, and thus minimize the chance of slow start overshoot.

There are four aspects of the FreeBSD implementation that can affect the performance of its gradient signal. Firstly, it prefers to select a positive g_{min} instead of g_{max} for backoff:

$$g_{backoff} = \begin{cases} g_{min} & \text{if } \textit{slow start} \text{ and } g_{min} > 0, \\ \hat{g}_{min} & \text{else if } \hat{g}_{min} > 0, \\ g_{max} & \text{else if } \textit{slow start} \text{ and } g_{max} > 0, \\ \hat{g}_{max} & \text{else if } g_{max} > 0. \end{cases}$$

This even applies in slow start, so that a smoothed but positive g_{min} is chosen over an unsmoothed g_{max}. Queue state is always determined by smoothed gradients, regardless of operating in slow start or not.

At the start of a connection when $n < w$, the window is zero-padded to account for $w - n$ missing gradients. The effect of this is noticed when exiting slow start before the window fills up.

The moving average is implemented using a fixed-point representation.[9] Integer gradients are scaled by 2^7 prior to insertion so that a decimal fraction of the summands are preserved after integer division by w. This limits the maximum window width to $2^7 = 128$ without distorting results.

Lastly, CDG uses measurements from the Enhanced RTT (Ertt) module [37]. This opt-in module provides the following enhancements [32]:

- Heuristically filters RTT measurements that are biased by delayed acknowledgements from the receiver. The algorithm detects whether the

[8] In a conference talk, Armitage suggested that the smoothed signal was needed for queue full estimation [4, 48–]. It is not a necessity for probabilistic backoff.

[9] Floating point representations are scarcely used in kernel code for reasons including: it is not universally supported by hardware in all target architectures (e.g. embedded), and supporting it in kernel code would typically require substantial overhead (memory and time) to save and restore FPU state when switching between kernel and user mode.

receiver employs delayed acknowledgements by counting ACKs that cover more than one segment. When detected, cumulative acknowledgement of single segments are presumed delayed and not used for RTT.

- Uses internal timing for measurements instead of TCP timestamps.[10] This enables RTT measurement for packets received out-of-order (selectively acknowledged), and mitigates the impact of receiver or middle-box tampering of TCP options. It is implemented by keeping a timestamped record of all transmitted segments and matching them against incoming acknowledgements. It may still use TCP timestamps, but only as an aid to validate the pairing of data and acknowledgement packets for internal timing.

- Intermittently disables TCP Segmentation Offloading[11] to provide accurate transmission timestamps once per RTT.

This module currently uses millisecond resolution for RTT measurements, which limits the range of detectable delays in each measurement to $\geq 1\,\text{ms}$.

2.6.2 Probabilistic backoff

Delay gradient signals indicative of congestion reduce the congestion window in a probabilistic manner. The immediate backoff probability is given by:

$$P_{backoff}(g) = 1 - e^{-g/G}, \qquad g \geq 0, \quad G > 0, \tag{2.6}$$

where g is g_{min} or g_{max} in milliseconds, and G is a scaling parameter (3 by default). The backoff probability is evaluated once per gradient calculation, i.e., at most once in a RTT.

With a steadily growing queue, a measured gradient grows proportionally to the RTT. The backoff probability increases exponentially with RTT in such a way that flows with long RTTs may have the same average probability of backoff as flows with short RTTs [34].

[10]TCP timestamps is a TCP header option with two 32 bit fields. Both ends set the timestamp (TS) field to a local clock upon transmission whilst the Echo Reply (ECR) field is set to reflect the latest sequentially received TS from the other end. The value of a received ECR can be compared against the current local clock to get an RTT measurement.

[11]TSO is a hardware offloading mechanism where the software stack queues a big chunk of data that the network hardware splits into smaller, separate packets for transmission.

2.6.3 Loss tolerance heuristic

The response to packet loss is conditioned on whether CDG presumes it to be correlated with congestion in the network. The desired effect is reducing spurious backoffs in environments that are not congested, but exhibit seemingly random packet loss due to other causes.

Congestion in the network is estimated using a state machine driven by the delay gradients g_{min} and g_{max}. State transitions are made using the following assumptions:

1. $g_{min} > 0$ and $g_{max} > 0$ indicates a *rising* queue, and loss due to congestion is imminent unless senders reduce their rate.

2. $g_{min} > 0$ and $g_{max} \leq 0$ indicates a *full* queue, and packet loss is due to congestion in the network.

3. $g_{min} < 0$ and $g_{max} < 0$ indicates a *falling* queue.

4. $g_{min} \geq 0$ and $g_{max} < 0$ indicates an *empty* queue, and the network is not congested.

Any other combination of g_{min} and g_{max} leaves the queue state unchanged.

Discussion

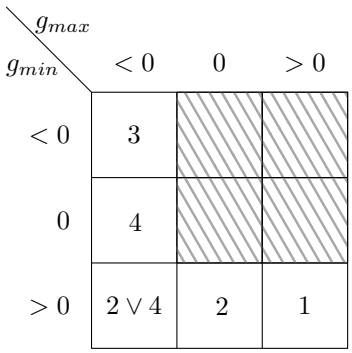


Figure 2.3: State transitions.

Figure 2.3 shows that there is an ambiguity between inferring a full queue (2) or an empty queue (4). The FreeBSD implementation gives precedence to inferring a full queue.

A potential weakness is estimating the queue state incorrectly when it is full [4, 01:08–]. The heuristic can ignore congestion-related losses unless the state machine inferred a full queue prior to detecting loss.

Delays are assumed to be dominated by a single bottleneck [4], although there could be several on the path between sender and receiver. It also assumes that tail drop is the only cause of congestion-related loss, i.e., it does not consider loss due to AQM mechanisms. These are limitations of the heuristic's design.

2.6.4 Competing with loss-based flows

Delay-based flows tend to cope poorly with competition from loss-based flows; they respond faster and back off more frequently because delay is a faster signal. CDG employs two mechanisms that helps it coexist in a mixed environment with loss-based flows.

Firstly, it employs a *shadow window* that enables it to undo delay gradient backoffs when packet loss occurs. The shadow window grows like the normal congestion window, but is untouched by delay gradient backoffs. It thus resembles the congestion window that the sender would have had if it ignored delay gradient signals. The shadow window s is updated at specific events:

$$s = \begin{cases} \max(cwnd, s) & \text{delay gradient backoff} \\ \max(cwnd, s)/2 & \text{loss backoff and full queue} \\ cwnd & \text{connection init} \\ cwnd & \text{congestion window validation} \\ s + 1 & s > 0 \text{ and } cwnd \text{ increased} \\ 0 & \text{empty queue} \\ s & \text{otherwise} \end{cases}, \tag{2.7}$$

as described in literature and the FreeBSD implementation [37, 33, 34].[12] The congestion window $cwnd$ is set to $\max(cwnd/2, s)$ on congestion-related packet loss. Since the shadow window is not penalized by delay gradient backoffs, it helps CDG compete with loss-based flows by retaining more of the congestion window when packet loss occurs.

Secondly, CDG detects and adjusts to ineffectual delay gradient backoffs. If delay gradients have been consistently rising after multiple delay gradient backoffs b, then these backoffs are assumed to be ineffectual because of competition with loss-based congestion control. Consequently, it ignores the next b' delay gradient backoffs (b and b' are 5 by default).

[12]References do not cater for congestion window validation. We include the case here for completeness.

2.7 Summary

This chapter presented the necessary background material for our work on delay-based congestion control. As a science, congestion control is a much larger interdisciplinary field that combines computer science, queueing theory, control theory, and others. Here, we merely focused on topics that guide and provide insight to Internet congestion control with special emphasis on TCP and network delays.

We summarized basic terminology of the Internet architecture and TCP in §2.1. We provided a concept of measuring the fairness of a congestion control in §2.2.1. We introduced the notion of a bottleneck in §2.3, why queueing is beneficial at bottlenecks in §2.3.1, some of the motivations for congestion control development in §2.3.2, how the management of queues influence network behavior in §2.3.3, and the concept of bandwidth in §2.3.4. We described a model for understanding packet delay in §2.4. We described how congestion controls can detect and react to congestion in §2.5. Finally, we described the design of the CDG congestion control in §2.6. In the next chapter, we examine our process of implementing CDG in Linux, and the efforts that we put into this implementation.

Chapter 3

Implementing CDG in Linux

> "I am really happy to see more congestion control development."
>
> — Stephen Hemminger, replying to our initial posting of CDG.

This chapter examines how we can implement new TCP congestion controls in the Linux kernel 4.1, and describes the efforts we put into pursuing our goal: having a working implementation of CDG as part of the Linux kernel.

3.1 Linux kernel development

The official repository for Linux' source code is available at http://www.kernel.org. For brevity, files referenced throughout this chapter are relative to the root of this repository. Development of networking functionality does not take place directly in this repository, but on branches of the kernel known as *net* or *net-next* that are currently maintained by David Miller.[1] Before making their way into the official kernel, bug fixes to existing networking code goes into net, and new networking functionality goes into net-next. We do not claim to be seasoned kernel developers, but the rest of this section provides a would-be kernel developer with some pointers for doing networking development.

Guidelines

Networking code is submitted as one or more patches to the network development mailing list, where they are subjected to peer review before potentially being committed by the maintainer. Patches submitted for inclusion in the

[1] His git repositories are available at https://git.kernel.org/cgit/linux/kernel/git/davem/.

kernel should follow its guidelines for coding style and conventions. The guidelines are currently part of the kernel sources as plaintext documents:

- `Documentation/CodingStyle` covers the base guidelines applicable to any kernel code.
- `Documentation/networking/netdev-FAQ.txt` makes additions for networking code, and also provides a more complete description of the review process for network development.

Conventions

The kernel has several general-purpose header files that provide macros and data structures for functionality such as atomics, lists, trees, and more. These are located under the `include/linux/` path. When applicable, their use is strongly encouraged in new kernel code to ensure similar code patterns across the kernel. Table 3.1 shows some of the data types and attributes that are defined by `include/linux/kernel.h`.

Debugging and instrumentation

Unlike user applications, kernel code can not access functions available in the standard C library, and should not output data directly to user terminals. It instead has other means available for doing debugging, instrumentation, and similar development tasks. An introduction to the kernel debugger or kernel profiler is not covered here; instead, we merely point out two logging facilities that can ease the transition from user mode to kernel mode development.

A convenient approach to obtain information about events in the kernel (from user mode) is to send messages to the computer's syslog facility using the `printf()`-style macros `pr_debug()`, `pr_warn()`, `pr_err()`, etc. However,

Type	Description
`s8/s16/s32/s64`	Signed integers having exact size of 8/. . . /64 bits.
`u8/u16/u32/u64`	Unsigned integers having exact size of 8/. . . /64 bits.
`__s8/__u8/...`	Variants that can be passed to user applications.
`__read_mostly`	A variable that has mostly read accesses (i.e. rarely writes).
`__pure`	A function that is mathematically pure (no side effects).

Table 3.1: Some of the data types and attributes used in kernel code.

the syslog is discouraged for debugging code with high event rates such as the networking stack. Calls to these functions can potentially cause delays or hangs that affect normal kernel operations. Linux provides a more suitable framework, *ftrace*, that has performant mechanisms to log debug messages, instrument CPU usage, and generate function call graphs. A drop-in replacement for `pr_debug()` is its `trace_printk()` function. When invoked, it stores a pointer to the provided format string and all its arguments as a binary blob in memory. This can be a denser storage format compared with plain text, and its memory-only approach obviates any time-consuming disk operations.

The ftrace buffer can be read in text form through the special file `/sys/kernel/debug/tracing/trace`. There is also a UNIX pipe interface that allows direct output to a terminal, accessible using the command `cat /sys/kernel/debug/tracing/trace_pipe`. Filtering mechanisms allow dynamically turning off and on different debugging statements as needed. Interested readers can consult `Documentation/trace/ftrace.txt` for more information. A downside of ftrace is that is requires support built into the kernel, and not all Linux distributions ship with this configuration. The less performant syslog facilities are widely available.

Assertion statements akin to `assert()` in the standard C library have a different interpretation for kernel code. In user mode, an assertion error typically kills the application; in kernel mode, an equivalent assertion either hangs or reboots the computer. Nonetheless, three common flavors of assertion statements are available with varying degrees of fatality: `BUG_ON()` either hangs or reboots the computer.[2] This is the only assertion that stops further code execution similar to `assert()`, and it is intended for truly critical errors that have no means of recovery. `BUILD_BUG_ON()` is a special assertion statement that is only evaluated during compilation, and it does not become part of the resulting machine code. It only works for statements that the compiler knows at compile-time, e.g., the memory size of a data type. `WARN_ON()` emits a warning in the syslog, but it does not inhibit further code execution. If in doubt, the preferred way to deal with runtime errors is to attempt recovery from the bad situation (e.g. clean up and return an error value), so that the kernel can resume normal operations.

[2] Depends on the setting of the `kernel.panic` sysctl. Set it to a value $n > 0$ for automated reboot after n seconds on kernel panic.

`if` expression	Interpretation
`before(a, b)`	if `a` is before `b`, then ...
`!after(a, b)`	if `a` is before or equal to `b`, then ...
`after(a, b)`	if `a` is after `b`, then ...
`!before(a, b)`	if `a` is after or equal to `b`, then ...
`a == b`	if `a` is equal to `b`, then ...

Table 3.2: Expressions for comparing TCP sequence numbers.

3.2 Congestion control development

Linux separates the operation of central TCP congestion control mechanisms into distinctive modules. Hemminger authored the original infrastructure that enabled such modules starting with kernel version 2.6.13 [38]. His proposal for a modular congestion control framework was motivated by:

- A desire to refactor Linux' support for multiple flavors of TCP congestion control. The flavors NewReno, Vegas, Westwood and BIC were at the time tangled into kernel networking code.
- Provide a means for developers to experiment with and add different congestion control mechanisms without further bloating the kernel.

As modules, congestion controls can be loaded, reloaded and unloaded without recompiling the kernel. The rest of this section describes common pieces of the TCP stack and how it fits into the congestion control framework.

3.2.1 Conventions

Macros and data structures specific to TCP are defined by `include/linux/tcp.h`. For brevity, we describe only the macros used in our CDG implementation:

The macro `tcp_is_cwnd_limited()` returns a boolean value that indicates whether the current connection is congestion window limited (`true`), or application-limited (`false`). In order to be congestion window limited, the connection must have utilized its entire congestion window in the previous RTT, i.e., the application must have sent sufficient amounts of data. Conversely, if the connection is application-limited, the application sent less data than it could have. Congestion window validation requires that the connection is congestion window limited when increasing the congestion window [31].

Hook name	Event/Purpose
`void cong_avoid`	cwnd: ACK or SACK of new data (required).
`u32 ssthresh`	cwnd: Loss or ECN congestion signal (required).
`u32 undo_cwnd`	cwnd: Undo reduction after a spurious loss signal.
`void in_ack_event`	ACK processing event.
`void pkts_acked`	ACK or SACK processing completed. ($\geq$ 4.1)
`void init`	Connection established.
`void release`	Connection terminated.
`void set_state`	Congestion control state transition.
`void cwnd_event`	Congestion control event.
`size_t get_info`	Request for debug info from user application.

Table 3.3: Function prototypes in `struct tcp_congestion_ops`. Functions that change the congestion window are denoted with cwnd. The first parameter is a pointer to socket state data (`struct sock *sk`).

Table 3.2 describes the conventional expressions for comparing TCP sequence numbers. The use of macros `before()` and `after()` in these expressions properly accounts for integer wrapping. TCP sequence numbers have a finite space of 32 bits, i.e., they range from 0 to $2^{32} - 1$. When a sequence number is increased beyond its maximum value, such as when a sender has transferred more bytes than a sequence number can represent, it wraps around and starts from zero due to the nature of integer arithmetic.

3.2.2 Programming interface

Congestion control modules implement the standardized callback functions in Table 3.3. They are required to implement `cong_avoid()` and `ssthresh()`, but the remaining callbacks are optional. Each of these callbacks provide a hook into the kernel networking code when specific events occur; these hooks allow a TCP congestion control to obtain information on network conditions and react accordingly.

Linux' TCP mechanisms may detect two event types that warrant reducing the congestion window: it detects a lost packet by means of a retransmission timeout or duplicate acknowledgement (loss signal), or it receives an acknowledgement with the Congestion Event Experienced flag (ECN signal). In either case, it invokes the congestion control module's `ssthresh()` callback

to obtain a new slow start threshold (ssthresh). As described in §2.2, the ssthresh should indicate the last rate (in MSS units) that is known to be sustainable by the network. In case of NewReno, this is calculated as the maximum of 2 MSS and half of the congestion window, i.e., $\max(2, cwnd/2)$.

Private memory area

Each established TCP connection currently reserves 64 bytes of zero-initialized memory for private use by congestion control modules. A void pointer to this memory area is obtained by calling `inet_csk_ca(sk)`, where `sk` is the first parameter to all callback functions. All congestion control modules in the kernel currently manage their private memory area by defining a custom data structure, and they use a check somewhere in their code that ensures this data structure is within the hard limit of 64 bytes (`ICSK_CA_PRIV_SIZE`):

```
BUILD_BUG_ON(sizeof(struct ...) > ICSK_CA_PRIV_SIZE);
```

Asserting this check at compile-time ensures that the module does not corrupt memory if the data structure inadvertently becomes larger than 64 bytes. Generic C types have a potential portability issue in this regard, e.g., the `long` type currently varies between a size of 32 bits or 64 bits, depending on compiler and architecture. `s32` or other types of exact size can be chosen to ensure consistent memory requirements.

3.2.3 Recipe for new modules

Figures 3.1 and 3.2 show our skeleton code to build a Linux congestion control module. Our skeleton merely demonstrates trivial use of the private memory area to count function invocations, and output of debug information via `trace_printk()` statements. It is akin to a "Hello World" program, but also achieves the required congestion control behavior by internal calls to NewReno's `tcp_reno_cong_avoid()` and `tcp_reno_ssthresh()`. Omitting required behavior could lead to congestion collapse as described in §2.1.

We save the skeleton code in a new directory, and compile the module by running `make`. After building it successfully, it can be loaded into the kernel:

```
root:~/example# insmod tcp_example.ko
```

Any software that supports pluggable congestion control is suitable for testing. As an example, we chose the iperf tool (v2.0.5) that is bundled with TEACUP. We start a server instance of iperf on our development machine using the command:

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <net/tcp.h>

struct example_priv {
        u32 call_count;
};

static void example_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
        struct example_priv *ca = inet_csk_ca(sk);
        struct tcp_sock *tp = tcp_sk(sk);

        ca->call_count++;

        tcp_reno_cong_avoid(sk, ack, acked);
        trace_printk("after %u calls, cwnd is %u\n",
                     ca->call_count, tp->snd_cwnd);
}

static u32 example_ssthresh(struct sock *sk)
{
        return tcp_reno_ssthresh(sk);
}

struct tcp_congestion_ops tcp_example __read_mostly = {
        .cong_avoid = example_cong_avoid,
        .ssthresh = example_ssthresh,
        .owner = THIS_MODULE,
        .name = "example",
};

static int __init example_register(void)
{
        BUILD_BUG_ON(sizeof(struct example_priv) > ICSK_CA_PRIV_SIZE);
        trace_printk("struct size: %zu bytes\n", sizeof(struct example_priv));
        tcp_register_congestion_control(&tcp_example);
        return 0;
}

static void __exit example_unregister(void)
{
        tcp_unregister_congestion_control(&tcp_example);
}

module_init(example_register);
module_exit(example_unregister);
MODULE_LICENSE("GPL");
```

Figure 3.1: `tcp_example.c` – CC module that piggybacks on NewReno.

```
obj-m := tcp_example.o

modules:
        make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
```

Figure 3.2: `Makefile` – builds `tcp_example.ko` for the running kernel.

```
root:~/example# iperf -s
```

We then start a client instance of iperf on the same machine using the command:

```
root:~/example# iperf -c localhost -Z example -l 1k -n 3k
```

We retrieve a plain text log from the ftrace buffer:

```
root:~/example# cat /sys/kernel/debug/tracing/trace
193.714496: example_register: struct size: 4 bytes
194.642222: example_cong_avoid: after 1 calls, cwnd is 10
194.642230: example_cong_avoid: after 2 calls, cwnd is 10
194.642233: example_cong_avoid: after 3 calls, cwnd is 10
194.642245: example_cong_avoid: after 4 calls, cwnd is 10
194.642387: example_cong_avoid: after 5 calls, cwnd is 10
```

3.3 RTT measurements

During our work to implement CDG in Linux, we explored how Linux performs RTT measurements and discovered shortcomings. In this section, we give a brief background to how Linux performs RTT measurements, and we succinctly describe the shortcomings that we found and relate them to kernel patches that we developed. One patch fixed bogus RTTs produced by retransmitted segments. The remaining four enabled RTT measurement from selective acknowledgements (SACKs). A full list of kernel patches was given in Table 1.1, and the full patch descriptions are in Appendix C.

Linux splits a TCP data stream into socket buffers (`struct skbuff`). These are inserted into the socket's retransmission queue, and timestamped upon transmission in `tcp_transmit_skb()` using `skb_mstamp_get()`. RTT measurements are produced in `tcp_clean_rtx_queue()` as the time between a segment's transmission and its acknowledgement; more specifically, the delta between current time and an acknowledged segment's transmission timestamp is given by `skb_mstamp_us_delta()`.

Retransmission

Socket buffers remain in the retransmission queue until they are cumulatively ACKed; more specifically, they remain there after being selectively ACKed. RTTs are computed using the latest acknowledged segment in the retransmission queue. Since retransmitted segments are never used for RTT due to ambiguity (Karn's algorithm), Linux inadvertently chose the latest non-retransmitted segment as depicted in Figure 3.3. One patch conditions

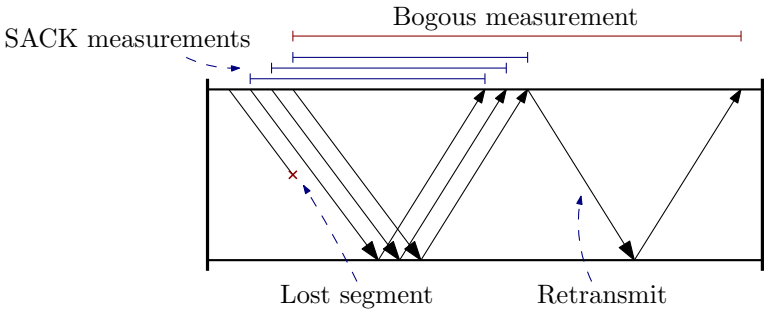


Figure 3.3: Bogous RTT measurement in Linux version ≤ 4.0.

measurements so that socket buffers are only used for RTT *once*, i.e., ACKing only a retransmitted segment does not produce any measurement.

Selective acknowledgement

Selective acknowledgements did not produce RTT measurements for congestion control. One patch selects the earliest SACKed segment for RTT, a measurement that is also used for estimating the retransmission timeout. Two later patches change data passing on the ACK code path so that measurements of the latest SACKed segment can be taken. This provides an improvement when SACKs are lost, or if the receiver delays SACKs in spite of RFC 5681 recommendations to send them immediately, e.g., ACK congestion control. The last patch passes pure SACK measurements to the congestion control module by invoking the `pkts_acked()` hook; previously, this hook was only called on the sequential (cumulative) acknowledgement of new data.

3.4 Implementation

Our Linux implementation is written from scratch to provide GPL-licensed code that takes advantage of Linux' latest congestion control infrastructure. We provided an independent implementation while honoring CDG's original design [34]. It has been inspired by previous implementations in FreeBSD and NS-2, and also bears similarities to other parts of Linux' networking code, e.g., `struct minmax` borrows the field reset approach used by `struct skb_mstamp`. The code in Appendix B was posted for review on the network development mailing list, and we received encouraging suggestions for improvements that need to be addressed before a final version can make its way into the Linux kernel.

3.4.1 Kernel implementation of $\exp(-x)$

The FreeBSD and NS-2 implementations use a lookup table with 641 elements to obtain values of the negative exponential function. Both tables provide a coarse representation of $\exp(-x)$, where $0 < x < 5$ in step increments of 2^{-7}.

We wanted an approach that could take advantage of Linux' more precise RTT measurements with microsecond granularity. We argue that a more accurate exponential function can achieve a more accurate backoff that is closer to CDG's intentional design, and that it would allow experimenting with a wider range of backoff scaling parameters.

The lookup table shared by FreeBSD and NS-2 implementations require $\approx 2.5\,\mathrm{KB}$ of memory. We could increase accuracy by adding more elements to the table, where the table size would then grow exponentially with accuracy. We also know that loading a missed cache line from main memory, such as when fetching a cluster of values in a lookup table, can be several orders of magnitude slower than a series of arithmetic operations in modern x86 CPUs [22]. We were thus motivated to explore possible alternatives that did not expand the lookup table.

Integer arithmetic is strongly preferred in kernel code. Our efforts to find an existing integer algorithm for the negative exponential function was unsuccessful using conventional means of literature searches and googling. We thus devised our own design based on the following identity of e:

$$e^{a+b+\cdots} = e^a \times e^b \times \ldots, \tag{3.1}$$

and the availability of computationally efficient methods to decompose an integer to powers of two by modern computer architectures. Our algorithm, shown in Figure 3.4, uses a small lookup table with 17 values of $\exp(-x \times 10^{-6})$, where $x \in \{0, 2^8, 2^9, \ldots, 2^{24}\}$. The lookup table requires a mere $2 \times 17 = 34$ bytes of memory, and the function is 82 bytes when compiled using a recent gcc in the kernel build environment. This design was devised to provide a fine-grained approximation for $\exp(-x)$ in step increments of 10^{-6}, i.e., in units of micro. This granularity is ideally suited for current RTT measurements in microseconds.

3.4.2 Backoff factor

The original design of CDG specifies a scaling parameter G to control the probability of backoff due to delay gradient indictions. We reformulate G as a *backoff factor* in our implementation, to allow for specifying fractional values of G using a single integer. We convert between them as:

$$backoff\_factor = 1000/G.$$

```
static u32 __pure nexp_u32(u32 ux)
{
    static const u16 v[] = {
        /* exp(-x)*65536-1 for x = 0, 0.000256, 0.000512, ... */
        65535,
        65518, 65501, 65468, 65401, 65267, 65001, 64470, 63422,
        61378, 57484, 50423, 38795, 22965, 8047,  987,   14,
    };
    u64 res;
    u32 msb = ux >> 8;
    int i;

    /* Cut off when ux >= 2^24 (actual result is <= 222/U32_MAX). */
    if (msb > U16_MAX)
        return 0;

    /* Scale first eight bits linearly: */
    res = U32_MAX - (ux & 0xff) * (U32_MAX / 1000000);

    /* Obtain e^(x + y + ...) by computing e^x * e^y * ...: */
    for (i = 1; msb; i++, msb >>= 1) {
        u64 y = v[i & -(msb & 1)] + 1ULL;

        res = (res * y) >> 16;
    }

    return (u32)res;
}
```

Figure 3.4: Our integer algorithm for the negative exponential function.

The default value of $G = 3$ thus has a backoff factor of ≈ 333. Our implementation allows for backoff factors ranging from 1 to 65535, which corresponds to scaling parameters ranging from $G = 1000$ to $G \approx 0.01526$. Conversely, conversion from *backoff_factor* to G is given by:

$$G = 1000/\mathit{backoff\_factor}.$$

A side effect of using *backoff_factor* as a replacement for G is that we can replace a division operation with a multiplication operation, i.e., we can replace $g/(1000 \times G)$ with $g \times \mathit{backoff\_factor}$, where g is a gradient in microseconds. Depending on the computer architecture, multiplication is likely a computationally cheaper operation than division, although we do not expect noticeable performance benefits from this small change on modern CPUs that support division in hardware.

3.4.3 Background traffic

As suggested to us during a discussion with Dr. David Hayes, we have added a module parameter, `use_shadow` (boolean), that allows the shadow window mechanism to be the disabled. This feature does not exist in the FreeBSD implementation. The ineffectual backoff heuristic can similarly be disabled by setting `ineffectual_thresh=0`, i.e., both coexistence heuristics are disabled by setting parameters `use_shadow=0 ineffectual_thresh=0`.

We have not run extensive experiments, but preliminary trials suggest that disabling both coexistence heuristics allows CDG to back off aggressively in the presence of competition from loss-based congestion controls. This would suggest that CDG could operate as a lower-than-best-effort congestion control for background traffic.

Previous experiments [5] have found that the existing FreeBSD implementation of CDG was a viable choice for background transport applications when a backoff beta parameter of 0.5 was used (default is 0.7), but they also found that CDG would retain some of its sending capability. Retaining some sending capability may, or may not, be beneficial for the particular applications in question, but a drawback of the change in backoff beta is the possibility that CDG may further undershoot its congestion window when doing a delay gradient backoff, whether competing or not. We propose that disabling both of CDG's coexistence heuristics, the ineffectual backoff detection and the shadow window, may inhibit its ability to grow the congestion window at any significant capacity in the presence of loss-based competition. It may reduce its congestion window close to the bare minimum of 2 MSS using the default backoff beta of 0.7, and thus it may also achieve a better performance in the absence of competition from other flows. However, further research is necessary to draw any conclusions.

3.4.4 Shadow window validation

Linux' TCP mechanisms gradually reduce the congestion window after brief periods of idle time, and discourages congestion controls from increasing the congestion window during application-limited periods. This is in compliance with the recommendations given for congestion window validation [31]. The purpose of prohibiting growth of *cwnd* during application-limited periods is to avoid invalid congestion windows; unless the congestion window is fully utilized, there are not enough packets in flight to determine whether the network supports the current congestion window or not.

In situations where we are application-limited, i.e., not using the full congestion window, we can not presume to know that the network would

support sending the shadow window either. Thus, whenever the connection is application-limited, we limit the shadow window to:

$$shadow\_wnd = \min(shadow\_wnd, cwnd).$$

After a sustained period of idle time, Linux may decrease the congestion window down to the Initial Window upon restarting transmission (currently 10 MSS in Linux). However, the connection is not necessarily application-limited when restarting transmission, e.g., it may transmit bursts of data at regular intervals. There may also be competition from loss-based flows, although the shadow window may have been empty when suspending transmission. Thus, in the event of restarting transmission, we re-initialize the shadow window as:

$$shadow\_wnd = cwnd,$$

to ensure that CDG is competitive against loss-based flows competing for capacity. This is in line with the FreeBSD implementation's behavior on connection establishment [37].

3.4.5 Proportional Rate Reduction

Linux' TCP mechanisms use Proportional Rate Reduction (PRR) [46] to gradually reduce the congestion window in response to a loss or ECN signal. Because PRR reduces *cwnd* in increments, rather than big jumps, it allows for a sender to keep sending (some) packets during the Congestion Window Reduced (CWR) state. Without PRR or a similar mechanism, reducing the congestion window below the current packets in flight would cease packet transmission completely until a sufficient amount of packets are acknowledged.

PRR helps improve fast recovery in the typical use case [46]. We consider PRR useful for backing off due to delay gradient indications because PRR helps keep the feedback loop going, i.e., it maintains the ACK clock. FreeBSD currently does not implement the PRR algorithm to perform congestion window reductions.

3.4.6 Loss tolerance heuristic

The design of CDG specifies a loss tolerance heuristic that aims to distinguish whether packet loss is due to congestion or not. It does so for the purpose of avoiding spurious reductions of the congestion window when packet losses are unrelated to congestion. As described in §2.6.3, this heuristic makes several assumptions about how congestion builds up in the network, and its decision to reduce the congestion window or not depends, respectively, on whether it infers a full queue in the network or not.

Erroneous inference of queue state

Our Linux implementation uses RTT measurements with microsecond precision, which is an improvement over the millisecond precision that Ertt provides to the FreeBSD implementation. However, this increased precision also opens for measuring the finer delays that Ertt ignores, such as the random variations in processing delays. In early experiments with our Linux implementation, we observed that the necessary condition for detecting a full queue ($\hat{g}_{min} > 0$ and $\hat{g}_{max} \leq 0$) was rarely inferred prior to congestion-related loss, and instead, the queue was estimated to be not full. The implication was that our implementation could achieve high throughput rates in steady-state when there was random loss, albeit because it effectively did not back off to any type of loss. This led to large performance variations between experiments, where CDG would vary between acceptable behavior, and periods of overly aggressive behavior with high congestion-related loss rates. We made a minor modification to alleviate this behavior, so that the queue state would reset to a conservative state on any type of loss, or on a delay gradient backoff. This ensured that, unless a new queue state had been inferred, successive losses would reduce the congestion window.

We also found that the smoothed gradients $\hat{g}_{min}$ and $\hat{g}_{max}$ would rarely reach zero, but instead fluctuate in the range of -4 to 4 microseconds. We believe that this effect could be due to minor variations that influence RTT measurements, e.g., processing delays, the intermittent queueing of ACK packets from the receiver to the sender, or inaccuracies in Linux' RTT measurements. The implication we found was that the necessary condition for detecting an empty queue (strictly $g_{min} = 0$ and $g_{max} < 0$ in FreeBSD, refer to Figure 2.3) was rarely inferred when the queue was actually empty. Because the state machine is particularly sensitive to conditions of $g > 0$, $g = 0$, or $g < 0$, our observation could also indicate that other states are misinterpreted. We suggest that rounding the gradients may help our implementation infer the queue state more accurately, although further research is needed to establish the appropriate amount of rounding. We did not evaluate the loss tolerance heuristic in the FreeBSD implementation.

Interaction with slow start

In the aforementioned experiments, we also observed that a major contributor to raising the congestion window above levels that the network could handle was the initial window probing in slow start. If delay gradient indications had failed to effect a backoff before filling the queue, the combination of a slow start overshoot and the loss tolerance heuristic would *preserve* a much too

high congestion window. When a loss was detected, Linux' TCP mechanisms would trigger a congestion window reduction (CWR) based on the *ssthresh* set by our Linux implementation. Because we (wrongly) inferred a non-full queue, we would set $ssthresh = cwnd$, i.e., no change. Once in the CWR state, Linux' TCP mechanisms would not trigger a new CWR episode for at least one RTT, and the delay gradient calculations would be suspended for two RTTs [34, §3.5]. Since the queue was still full after exiting CWR, a successive packet loss would immediately trigger a new CWR, and the process repeated. The cycle was eventually broken because socket buffers got exhausted, i.e., no more data could be transmitted out-of-order. Once the queue drained, gradient calculations could be performed, and adequate performance was achieved due to delay gradient backoffs. Although our aforementioned modification of resetting the queue state on loss may alleviate this problem sufficiently, future work may explore whether intermittently disabling the loss tolerance heuristic in initial slow start could provide a further improvement.[3] We have not explored the implications of this latter suggestion, and opted not to make this change in our implementation.

Implementation considerations

We made a small change to the loss tolerance heuristic's design that reduces the impact of an erroneous inference. We point out that the particular problem of slow start overshoot may be alleviated by use of the Hybrid Slow start algorithm [29, 21], and suggest implementing Hybrid Slow start in CDG as an area of future work.

As we claimed in §2.5.2, delay can best be understood as a supplementary congestion signal. The implication is that we cannot assume a reliable delay signal in the Internet. Similar to original CDG experiments, we used a bottleneck that shaped bandwidth to 10 Mbps, where packet delay is correlated with congestion. Conversely, this is not necessarily true for bottlenecks that use policing. Should CDG fail to back off sufficiently due to delay gradient indications, the combined effect with its loss tolerance heuristic could undermine the purpose of congestion control. In the event that an ECN signal is received, we ensure that the loss tolerance heuristic is intermittently disabled.

We suggest that future research is necessary to ensure that the loss tolerance heuristic is safe for use in the Internet. The loss tolerance heuristic may hold merit for use in certain closed environments, for example:

[3] Linux provides a macro, `tcp_in_initial_slowstart()`, to determine whether the connection is in initial slow start or not.

- where ECN is deployed in the network and activated at end hosts, or
- where delay as a congestion signal can be assumed to be reliable, and the fundamental assumption of the heuristic's design holds, namely the use of a tail drop queue.

We have added a module parameter, `use_tolerance`, to disable or enable this heuristic. Since we cannot conclude that it is safe for use in the Internet, it is disabled by default in our implementation.

Summary

In this chapter, we described aspects of Linux kernel development with an emphasis on networking. We described some of the discoveries we made regarding Linux' RTT measurements, and our contributions to improve them for all delay-based congestion controls in the Linux kernel. We described some parts of our Linux implementation of CDG that makes it different from the FreeBSD implementation, and we described some of the issues that we encountered during development of our Linux implementation. In the next chapter, we will describe the evaluation of our final CDG implementation, including some of the efforts we made to ensure valid experiment conditions, and to seek out hidden variables that influenced the performance of implementations.

Chapter 4

Evaluation

When evaluating a congestion control there are many considerations, including how fairly it can compete with instances of itself or other congestion controls, and how its performance varies in a vast range of diverse environments [25, 24, 35]. In this section, we describe our limited experimental evaluation of the CDG congestion control for the purpose of comparing our Linux implementation with the reference implementation in FreeBSD.

4.1 Experiment testbed

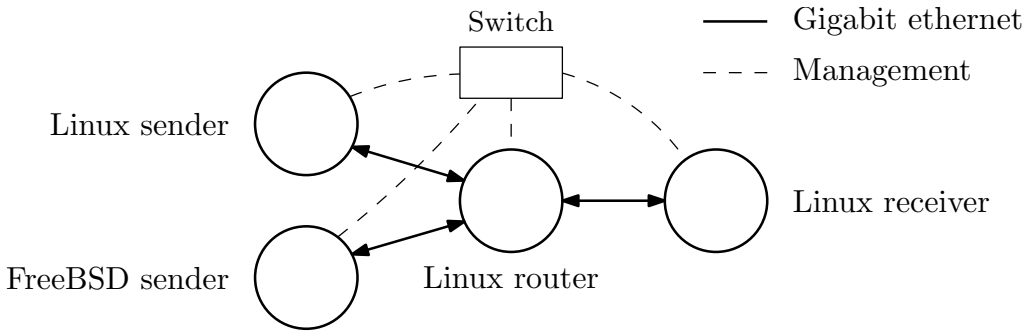


Figure 4.1: Testbed topology

Figure 4.1 depicts the network topology used for testbed experiments. All nodes are connected to a shared switch for management traffic. For experiment traffic, we used a star topology with every end host connected to one Linux router. Each end host has a separate IPv4 subnet to ensure that only the traffic traversing end-to-end is affected by the bandwidth limitation. Necessary broadcast traffic ($\leq 1Kbps$) is thus isolated to the router-host

interface. Hardware and operating system details for the testbed nodes are described in Appendix D.

4.1.1 Data gathering

Our experiments used the TEACUP framework [74] (TCP Experiment Automation Using Python). Given a configuration file with experiment conditions, it sets up network parameters, data collectors and executes test scenarios. Using version 0.8.1 as a base, we made some changes to its operation:

Hardware timestamps Intel network adapters in our Linux nodes support hardware timestamping of both received and transmitted packets. This provides us with timings that are unbiased by operating system noise, given only that the system is configured to use this feature. We reconfigured our kernel, libpcap and tcpdump for support, and made an adjustment to TEACUP's data collection routine: invocations of tcpdump must use the parameter `-j adapter`, which specifies the use of adapter timestamps for recording packet traces.

RTT analysis Synthetic Packet Pairing (SPP) is a tool bundled with TEACUP, and we used it for offline RTT analysis from packet traces. In SPP terminology, TEACUP uses a fixed setup where the receiver host is the *monitor point*, and the sender host is the *reference point*. SPP produced per-flow CSV-files containing two fields for each pair of data and ACK: a receiver timestamp, and the estimated RTT. We experienced a number of limitations and caveats with SPP, and TEACUP's invocation of SPP:

- SPP uses receiver packet traces to estimate *receiver processing time*, e.g., accounting for delayed acknowledgement timeout. This time is then subtracted from the RTT seen by the sender [73], i.e., SPP estimates an RTT as:

$$\begin{aligned} rtt &= t_{ack\,received} - t_{data\,sent} - processing, \\ processing &= t_{ack\,sent} - t_{data\,received}. \end{aligned}$$

  Measurements in Linux or Ertt do not account for the receiver processing time. To compare them fairly with SPP's estimates, we enabled a feature that exports the receiver processing time.

- We found that SPP can distort the experiment timeline. Code review (`spp-0.3.5/src/pair.c`) revealed that SPP's default behavior is to estimate timestamps at the receiver using extrapolation:

$$t_{receiver} = t_{ack\,received} - (rtt + processing)/2, \qquad (4.1)$$

where $t_{ack\,received}$ is a timestamp at the sender. This timestamp is incorrectly estimated when one-way delays are different; in our experiments, this was evident when the path from the sender to the receiver was congested.

- SPP's design is generalized for RTT estimation in the network, and does not make assumptions about clock synchronization at end hosts [73]. However, in our experiments, we can assume that the clock drift between the sender and the receiver is approximately constant for the short time each experiment runs. This enables us to a establish time offsets in experiment data.[1] We changed the code to export two real timestamps: departure time at the sender, and arrival time at the receiver. The former time is synchronized with congestion control events at the sender, and the latter is synchronized with calculated flow throughput at the receiver.
- We added code to extract auxiliary information that uniquely identifies the pairing of segments and ACKs used for RTT estimates. This allowed SPP's estimates to be compared with measurements produced by Linux and Ertt.

Congestion control data Linux provides a general-purpose tcpprobe module for recording changes in common congestion control variables. However, its design does not cater for recording data and events private to a congestion control module. We used Linux' ftrace framework to collect additional information from within our CDG module:

- Delay gradient backoffs, and non-backoffs.
- Changes to *cwnd, ssthresh,* and shadow window.
- RTT measurements, filtered measurements, and gradients.
- Auxiliary information that uniquely identifies the connection, and the last acknowledgement number received. This was used for pairing with the RTT from SPP.

Data was stored in ftrace's memory buffer during experiments, and dumped to files after experiments ended. We used debug logging facilities for similar data in FreeBSD.

[1] Time offsets are particularly accurate between Linux hosts due to their use of hardware timestamping by the network adapter.

4.1.2 Issues and mitigations

We identified several hidden noise variables that can affect the performance of delay-based congestion control in our experiment setup. We opted to eliminate some of them when possible, and do not thoroughly evaluate their impact in this thesis:

Switch processing Hosts were originally interconnected via a switch, using VLAN-tagging to separate experiment and management traffic. As measured by the ping tool (part of iputils), the switch was found to introduce delay on the scale of several hundred microseconds. We further made a small modification to help ping utilize hardware timestamps from the network adapter, but without observing improvement in delay jitter. We could not explain this jitter from traffic volume, but propose that it is related to processing delays inside the switch. Jitter reduced to $\leq 20\,\mu s$ after removing the switch (50000 packets in both cases). This discovery warranted use of dedicated Ethernet interfaces for experiment traffic between the router and hosts.

Flow control Ethernet flow control is supported by all nodes in our testbed. Our experiments build queues in the direction towards destination, and we are not aware of any routing mechanism that uses flow control to assert back pressure on the source in this scenario, e.g., the router used for bottleneck simulations has no known incentives to exert flow control throttles due to congestion. However, flow control still poses a potential risk to bias sender behavior; we disabled support on Linux nodes using the command `ethtool -A ethX autoneg off tx off rx off`.

Adapter buffering in Linux Our Linux sender, router, and receiver use Intel network adapters with the e1000e driver. By default, these adapters impose a delay on receive and transmit interrupts to lower CPU load. We disabled both features by setting driver module parameters `InterruptThrottleRate`, `RxIntDelay`, and `TxIntDelay` to 0 for all adapters. Note that these parameters take an array of integers, one value for each network adapter.

Adapter buffering in FreeBSD Our FreeBSD sender similarly has a network adapter that delays interrupts. The `msk` driver operates using a maximum interrupt delay of 100 µs by default. We disabled interrupt delays by configuring the sysctl `dev.mskc.0.int_holdoff` to 0.

Hardware offloading The TEACUP framework used in our experiments disables a hardware offloading mechanism known as General Segmen-

tation Offloading (GSO) in Linux. This can improve the delay characteristics of routing and switching by reducing per-packet processing delay, albeit at the cost of more CPU overhead. A similar mechanism known as TCP Segmentation Offloading (TSO) is supported by both Linux and FreeBSD, and can reduce per-segment processing delay in a similar fashion. These platforms implement it slightly differently, and employ different heuristics to circumvent buffer bloat or spurious RTT measurements, e.g., the Enhanced RTT module disables offloading for one packet per RTT. In addition to ruling out a noise variable, we were motivated to disable TSO at hosts for fair comparison of the congestion control implementations.

4.2 Metrics

Metrics provide us with a method to evaluate quantitive information. The choice of metrics is important for understanding the impact of the mechanisms that we study. For our case of evaluating delay-based congestion control, we have used the following metrics:

SPP RTT The round-trip time estimated by offline analysis using a modified version of the SPP tool. It is the time between receiving acknowledgement and sending the last segment covered by that acknowledgement.

Queueing delay The queueing delay measured by the sender and the receiver in cooperation. Reconstructed from pcap-files. We calculate the offset between sender and receiver timestamps, then subtract the minimum offset observed for each group of packet size and experiment run. We group by packet size to account for serialization delay, and by experiment run to account for small drifts in clock offsets. We assume at least one packet transits an initially empty queue in each experiment run, thus providing the ideal minimum one-way delay.

CDG RTT The round-trip time as seen by the congestion control. Estimated by the Linux kernel, or the Enhanced RTT module in FreeBSD.

Backoff probability The number of backoffs divided by the number of possible backoffs. We include negative gradients as a "possible backoff" in this calculation to account for any difference in gradient distributions between implementations.

Throughput The network layer data successfully transmitted from sender to receiver measured as flow rate in bits per second. It does not

include dropped packets, or overhead at layers below the network layer. Calculated from packet traces at discrete 1 s intervals.

Fairness Jain's fairness index, as described in §2.2.1. Based on Throughput at discrete 1 s intervals.

CDG RTT and Backoff probability is obtained by use of ftrace in Linux, or the debug logging facility in FreeBSD.

4.3 Homogeneous capacity sharing

We have, to our knowledge, recreated the experiment scenario "Homogeneous capacity sharing" that is described by Hayes et al. [34] This scenario is designed to assess how CDG competes with instances of itself, and we are inclined to recreate it for the purpose of comparing our results with the results that they have reported. We have also used additional parameter settings to observe a greater extent of CDG's behavior.

We ran three greedy flows for 60 s, all using the iperf tool configured with a socket buffer of 1.5 MB and the CDG congestion control. Flows started at $t = 0$ s , 20 s, and 40 s. The bottleneck bandwidth was shaped to 10 Mbps, while link capacities were 1 Gbps between all hosts. The queue length was limited to 84 packets using tail drop. We artificially added symmetric propagation delays of 0 ms, 10 ms, 20 ms and 35 ms. We used the scaling parameters $G \in \{1, 2, 3\}$. Each combination of parameters was repeated for 30 runs.

Compared with the original experiment, we repeated each parameter setting for 30 runs instead of 10. We also added propagation delays 0 ms, 10 ms, and the scaling parameters $G \in \{1, 2\}$. However, we do not present the results for inducing one percent random loss that the original experiment used to evaluate CDG's ability to infer whether losses are related to congestion. As we described in §3.4.6, the original design of the loss tolerance heuristic did not perform adequately in our experiments, and we argued that design changes may be necessary for it to be useful in our Linux implementation. As there was not enough time to design, implement and evaluate such changes, this was categorized as future work.

Performance comparison

Figure 4.2 shows box-plot statistics of the summed throughput for all three flows. Throughput appears to be equivalent for $Delay = 0$, but we observed that the Linux and FreeBSD groups differ in all other cases. The median

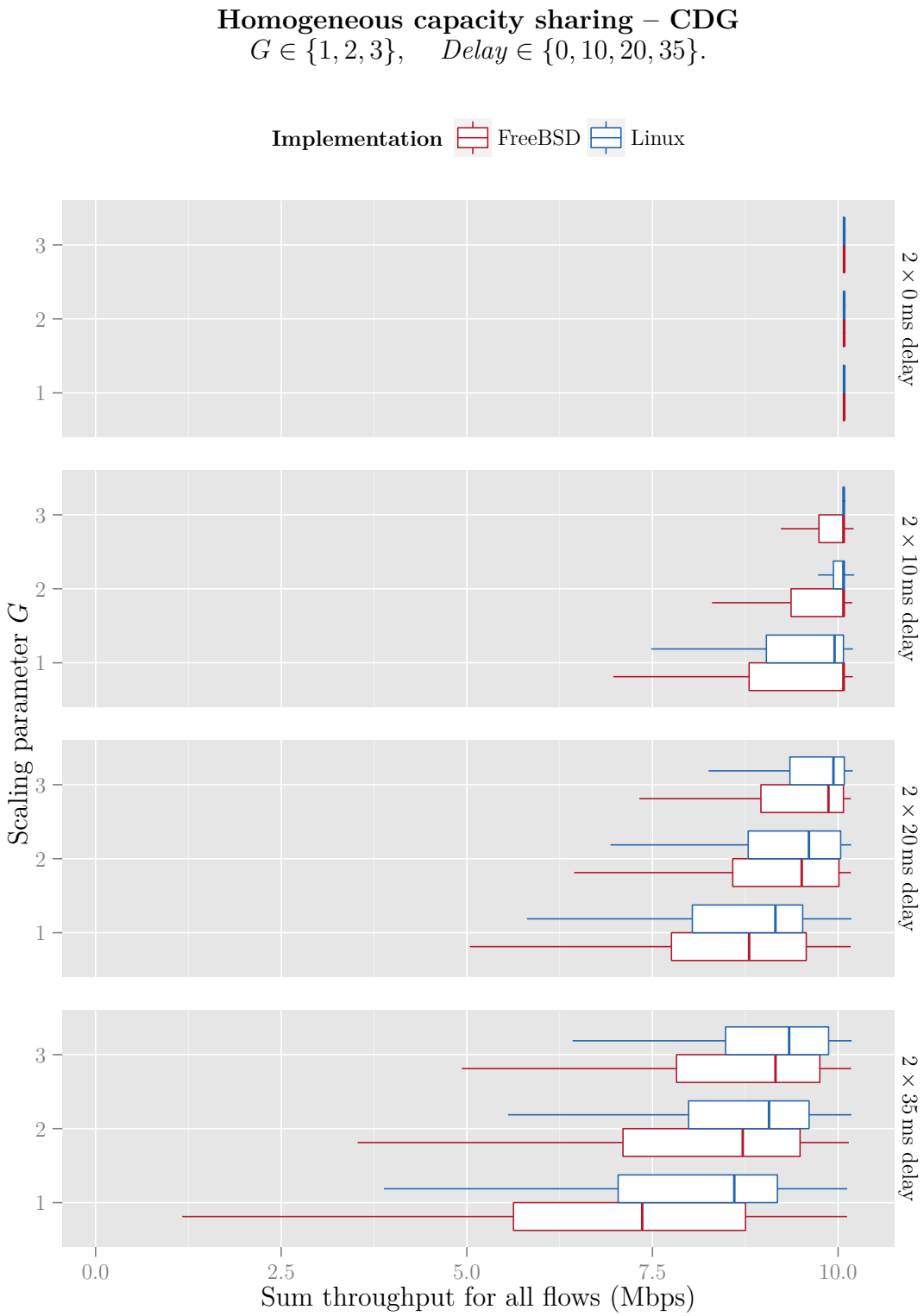


Figure 4.2: Throughput for all three flows, summed at discrete 1 s intervals. Box-plot statistics with outliers removed due to a processing limitation.

throughput achieved by Linux outperforms FreeBSD for $Delay \in \{20, 35\}$. To better understand the subtle mechanisms behind the performance difference in implementations, we observed the data presented as a time series; since each combination of delay and the scaling parameter G requires a distinct time series for fair representation, we chose to present $G = 3$ and $Delay = 35$ here, and include the remaining data in Appendix A.

Figure 4.3 and 4.4 shows the achieved throughput, fairness and backoff probability as a time series for Linux and FreeBSD groups, respectively. We observe that FreeBSD attained fairer throughput allocations in the competing region ($20 \leq t < 80$), although Linux attained a better throughput in the absence of competition ($t < 20$ and $t \geq 80$). Another observation is that the backoff probabilities were slightly higher for FreeBSD, shown here through the median and third quantile. From exploring the design of CDG, we knew that the Enhanced RTT module used in the FreeBSD implementation produced measurements of millisecond precision. Our initial hypothesis to the probable cause was that a combination of Linux' more precise RTT measurements and its more precise function for $\exp(-x)$ led to more precise backoffs. We explored this hypothesis by:

- Replacing the more precise $\exp(-x)$ with the coarser lookup table algorithm used in the FreeBSD implementation.

- Retrofitting millisecond precision into the Linux kernel, providing both µs and ms precision RTTs to congestion control modules. A kernel-level implementation was necessary in order to replicate the exact approach that the Enhanced RTT module uses to produce measurements.

Neither of these avenues produced observable effects for the given experiment parameters (i.e. Homogeneous capacity sharing).

Figure 4.5 shows the empirical RTT distributions for all experiment variables. We observed that the FreeBSD group had a larger accumulation of low RTTs compared with the Linux group. Our earlier efforts to explain the throughput difference based on accuracy of backoff and precision of RTT measurements could not explain why FreeBSD had this distribution. Our second hypothesis to explain the difference in performance was that the FreeBSD implementation had some element of random backoff that was not based on the actual RTT signal, e.g., a hidden noise variable that could cause oscillations in RTT measurements, yielding a larger number of positive gradients, and thus inducing a certain level of backoffs at random. We explored this hypothesis by analytical and experimental evaluation into the sources of noise in the Enhanced RTT module; we did find sources of noise,

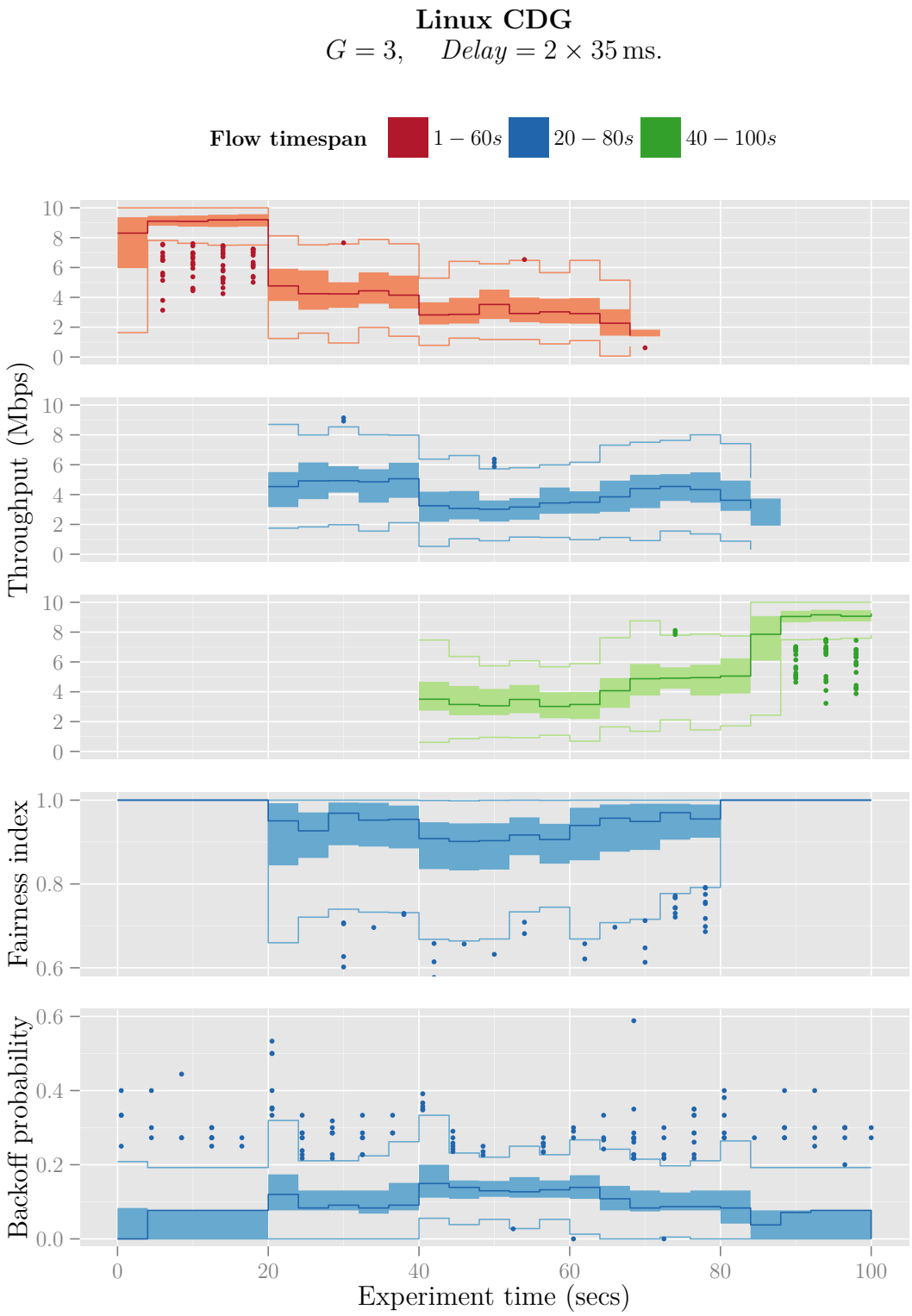


Figure 4.3: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

but our findings did not yield conclusive results for their impact on CDG's backoff probability.

Our third hypothesis to explain the performance difference is that the use of Proportional Rate Reduction (PRR) in our Linux implementation is the primary contributor. We argue that this is a possible explanation, but we do not currently have empirical results to make conclusions on the matter.

Throughput and flow rate fairness

In Figure 4.2, we present box-plot statistics that compared the throughput obtained by Linux and FreeBSD implementations. We observed a trend in this figure, where smaller values of G obtain less throughput. This is to be expected from the nature of CDG's backoff function, where larger values of G lead to a lower probability of backoff.

We also observed that Linux mostly outperformed FreeBSD, which could suggest that the Linux implementation is capable of obtaining a throughput that is closer to a NewReno flow than the reference FreeBSD implementation. In §3.4.5, we described that an advantage of Proportional Rate Reduction (PRR) is that it allows the sender to keep some segments in the network during a congestion window reduction even if the congestion window target is below the current packets in flight. PRR has a smoothed and gradual effect on backoff that potentially keeps the link occupied for larger amounts of time. This could be advantageous, e.g., in cases where a single flow is dominating the link, and a backoff due to delay gradient indications leads to a draining of the bottleneck queue. We argue that PRR could be the primary contributor to CDG's increased throughput in Linux, but we do not currently have empirical results to make conclusions on the matter.

Figure 4.6 shows box-plot statistics for flow rate fairness with varying parameters of *Delay* and G. The measured fairness suggests that FreeBSD outperforms Linux in terms of achieving flow rate fairness. One possibility is that the FreeBSD implementation has a larger degree of randomness in its backoff probability. Another possibility could be that the Linux implementation suffers more from phase effects than the FreeBSD implementation. It is possible that PRR induces phase effects in homogeneous CDG environments, e.g., since it keeps more packets in the network during a congestion window reduction, other flows may still detect a rising queue after one flow backs off. However, we do not currently have empirical results to make conclusions on the matter, and suggest that future work explore this phenomenon.

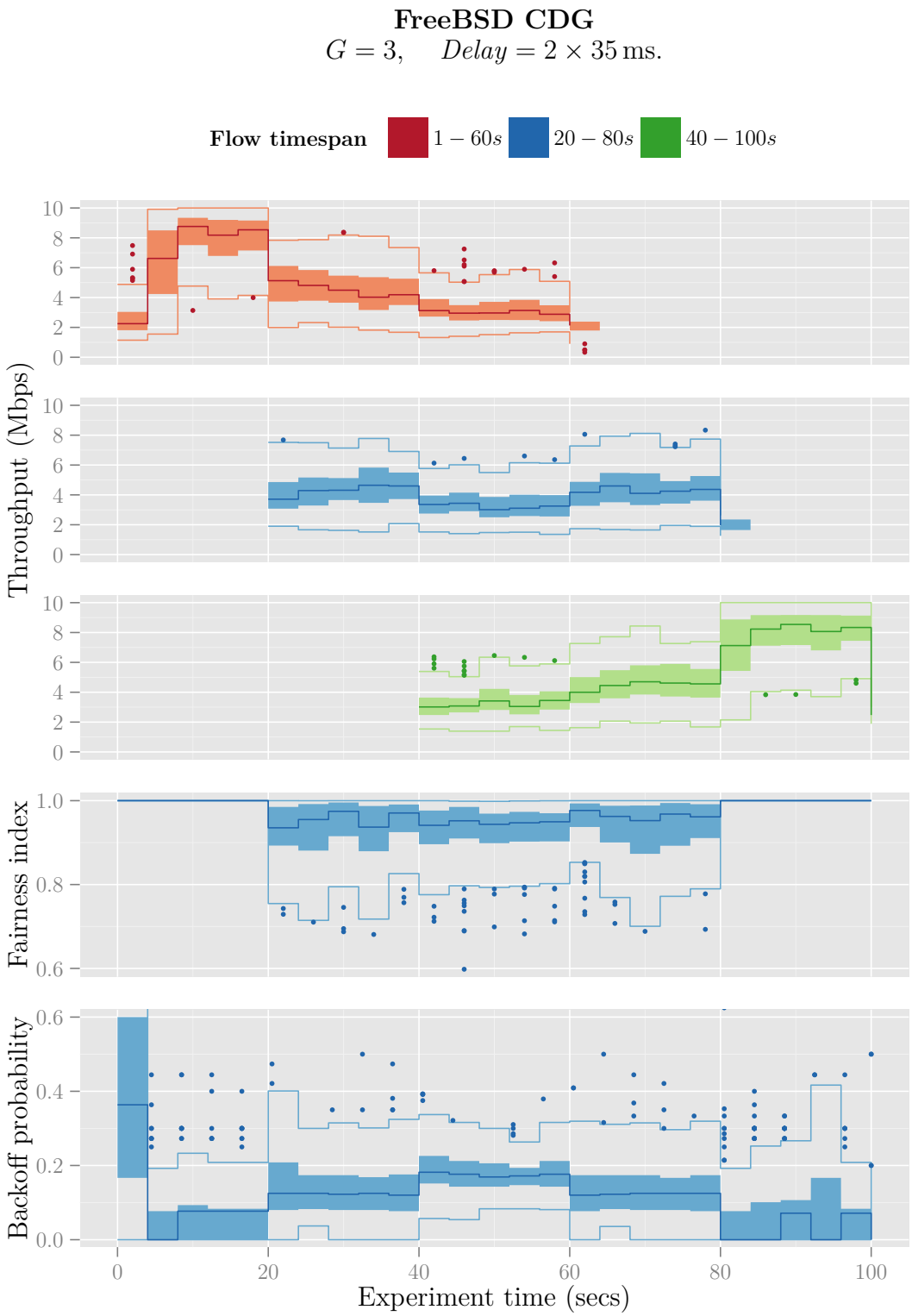


Figure 4.4: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

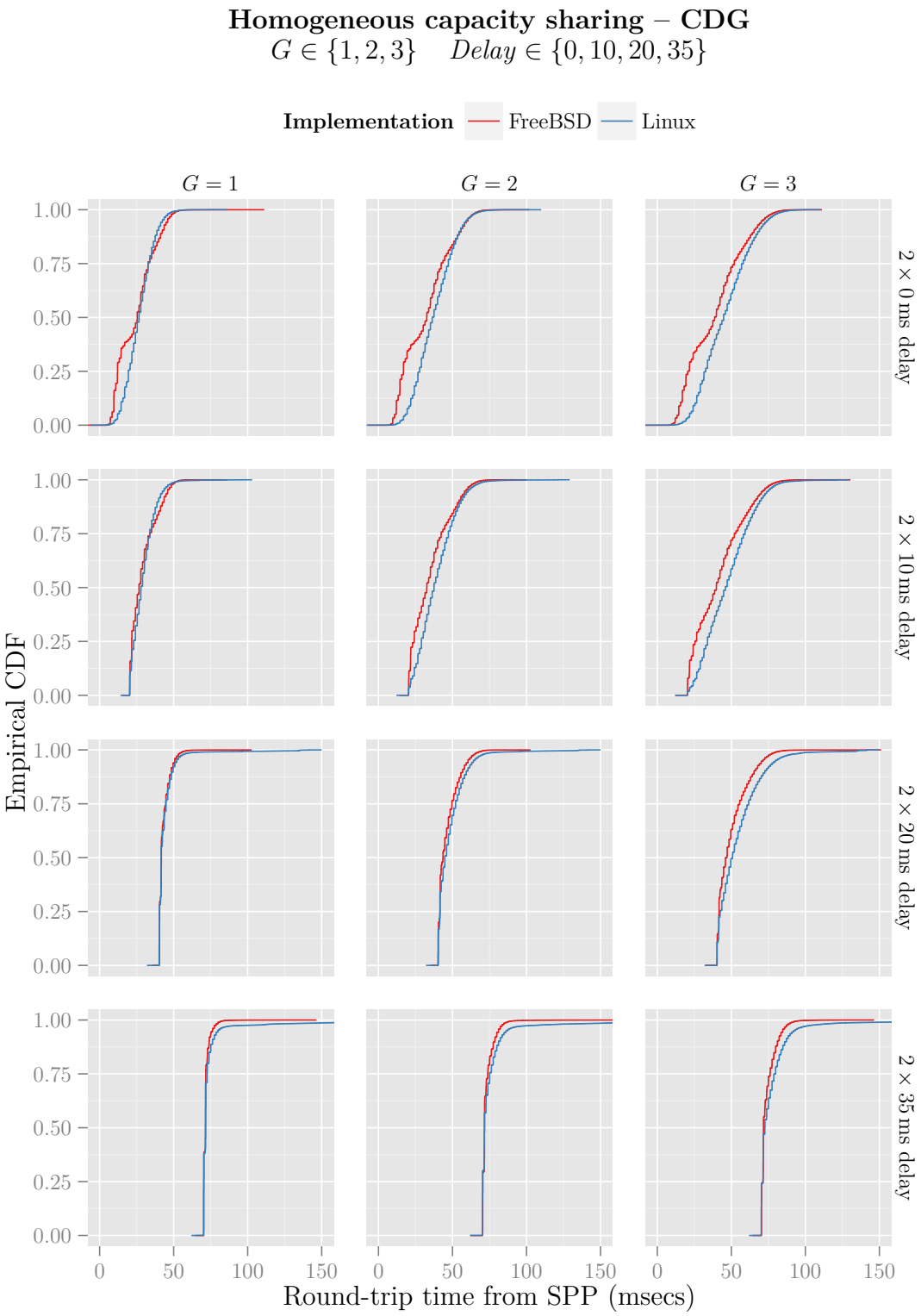


Figure 4.5: RTT distributions, as estimated by SPP. Receiver processing time is not included.

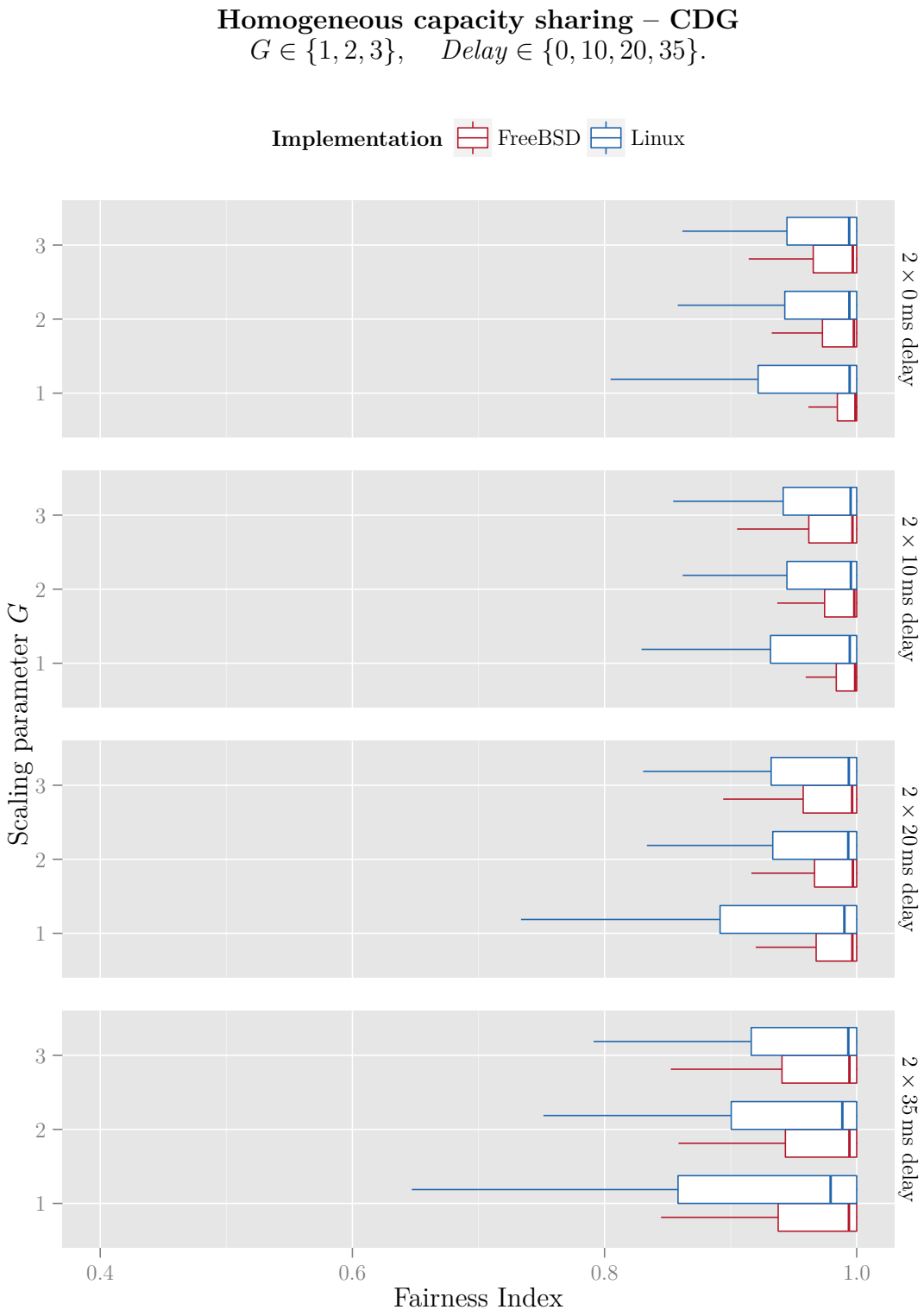


Figure 4.6: Flow rate fairness for CDG calculated at discrete 1 s intervals. Box-plot statistics with outliers removed due to a processing limitation.

RTT dependency

In Figure 4.5, we presented the RTT distributions, where the the delays induced by CDG diminished as the propagation delay increased. Figure 4.7 more clearly shows the queueing delay that was induced by CDG. We suggest that these trends in queueing delay are indicative of an RTT dependency, where the probability of underutilizing the bottleneck queue increases with RTT. This is supported by throughput statistics in figure 4.2, where throughput decreased with increases in RTT.

The delay gradient backoff function employed by CDG aims to be independent of RTT. However, the AIMD mechanisms of NewReno that CDG inherits are very much dependent on the base RTT, relying on feedback from the receiver to grow the congestion window. The CUBIC window increase algorithm is designed to be independent of base RTT [30]; we suggest that future work explore an implementation of the CUBIC window increase algorithm as a replacement for NewReno in CDG.

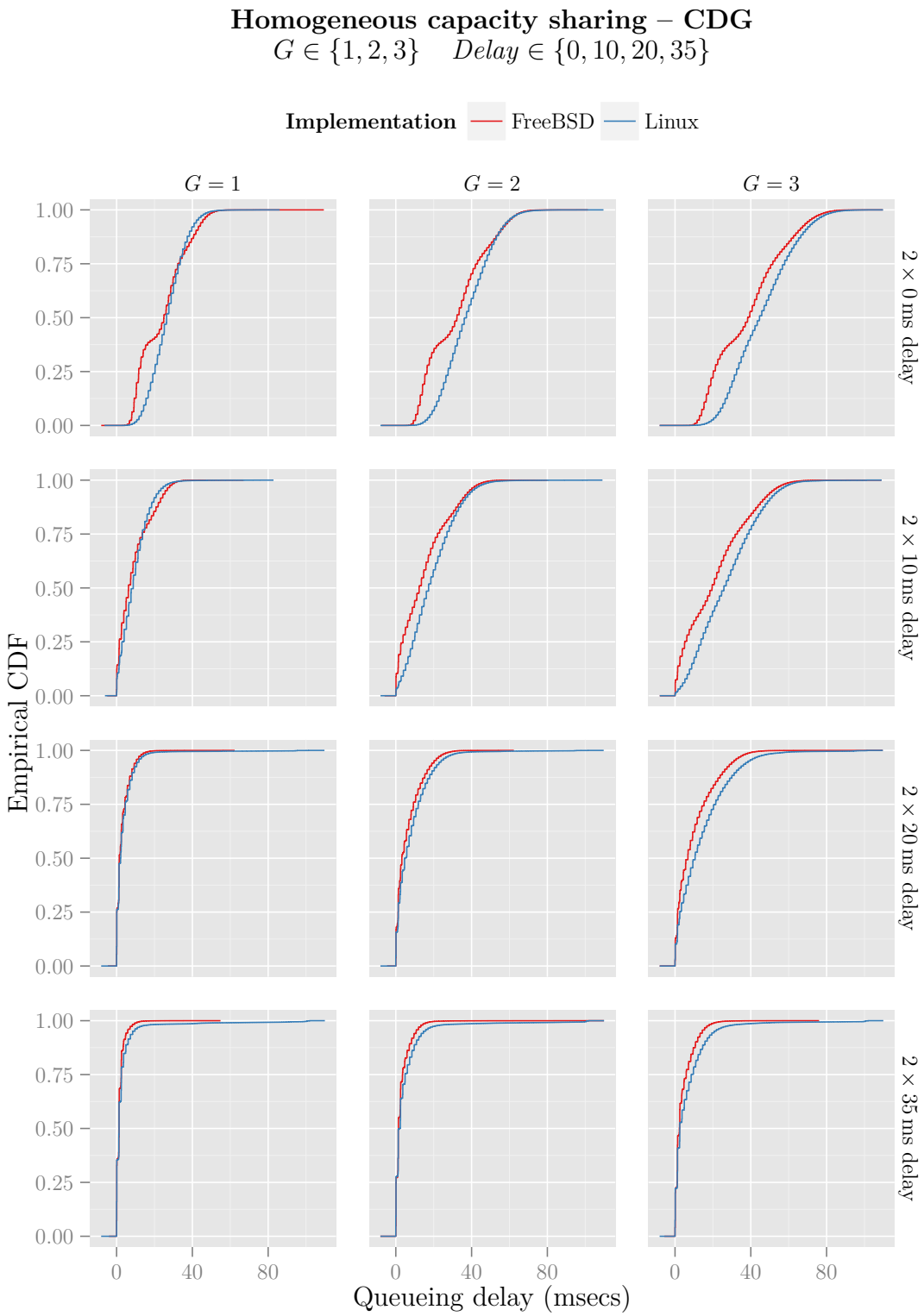


Figure 4.7: Queueing delay distributions, as measured by the sender and the receiver in cooperation.

4.4 Enhanced RTT module

We have reviewed the source code and operation of the Enhanced RTT module (Ertt) used by CDG's FreeBSD implementation. This was motivated by a desire to explain the observed performance difference between FreeBSD and Linux implementations of CDG. At time of writing, our observations correspond with source code in the current development branch of FreeBSD.

The Enhanced RTT module records each segment's transmission time tx_ts, and produces RTT measurements $rttm$ when receiving acknowledgements:

$$\begin{aligned} tx\_ts_i &\leftarrow \text{tcp\_ts\_getticks}(), \\ &\vdots \\ rttm_i &\leftarrow \text{tcp\_ts\_getticks}() - tx\_ts_i + 1, \end{aligned} \tag{4.2}$$

where the addition of a constant 1 assures that no valid measurement is underestimated or zero.[2] There is a logic for matching acknowledgements to their corresponding transmission time tx_ts which we leave out for brevity. Our code review suggests that the logic is correct, and we have no experimental observations that indicate otherwise. This lends support to prior reports that the module correctly accounts for retransmission ambiguity and selective acknowledgements [32].

Precision and accuracy

`tcp_ts_getticks()` produces a single integer by means of acquiring system uptime in microseconds and reducing its precision to milliseconds:

$$(uptime_{sec}1000)\,\text{ms} + \lfloor uptime_{usec}/1000 \rfloor\,\text{ms}, \tag{4.3}$$

where flooring follows from integer devision. It follows that resulting RTT measurements have a precision in the range of milliseconds.

We can establish measurement accuracy given that system uptime is monotonic and accurate.

Case 1. Assume a segment is transmitted at 0 µs and its acknowledgement received at 999 µs with true measurement of $rtt = 999\,\mu\text{s} - 0\,\mu\text{s} = 999\,\mu\text{s}$. Using 4.2 gives a measurement of $rttm = 0\,\text{ms} - 0\,\text{ms} + 1 = 1\,\text{ms}$ and error of $rttm - rtt = 1\,\mu\text{s}$.

[2] A variable value of zero conventionally indicates that no value is set.

Case 2. Assume a segment is transmitted at 999 µs, and its acknowledgement received at 1000 µs with true measurement of $rtt = 1000\,\mu s - 999\,\mu s = 1\,\mu s$. Similarly, we get a measurement of $rttm = 1\,ms - 0\,ms + 1 = 2\,ms$ and error of $rttm - rtt = 1999\,\mu s$.

From these two cases, we have a lower and upper bound for the accuracy:

$$0\,\mu s < rttm - rtt < 2000\,\mu s. \tag{4.4}$$

Experimental evaluation of measurement noise

Our bound made certain non-trivial assumptions about the clock source:

Monotonicity Ensures that time does not reverse itself. If uptime is calculated relative to a wall clock, monotonicity can be broken by NTP updates or other means of adjustment.

Accuracy Ensures that time progresses at an even and accurate pace. If uptime is calculated from a wall clock, then leap seconds break accuracy. If uptime is calculated from the number of CPU cycles since boot, then events such as system sleep, hibernation or virtualization can also break this property.

We use an experimental approach to validate that our analytical assessment is correct, and to quantify and evaluate the uncertain parameters in our assumptions. Experiments replicated the scenario in §4.3, with 6×10 repeated runs that vary parameters $G \in \{2, 3\}$ and $Delay \in \{10, 20, 35\}$. In addition to the data collected by TEACUP, we modified the Ertt module to log some of its internal data. For each pair of packets that produce Ertt's RTT measurements, we logged the following:

- Full microsecond values returned by `getmicrouptime()` for segment transmission tx_1 and acknowledgement ack_1. This clock is the source for Ertt's measurements, but we capture it before Function 4.3 truncates it to millisecond precision.

- Similar values tx_2 and ack_2 returned by `microuptime()`. The former uptime function is a coarser variant of the latter. Calls to `microuptime()` were made prior to `getmicrouptime()`, ensuring that we logged the latest possible value from `getmicrotime()`. As we are interested in obtaining the noise caused by the coarser function, this ordering rules out any occurrence of false positives.

- Auxiliary information that allowed RTT measurements from Ertt to be compared with RTT estimates from SPP, and a counter that incremented for each calculation of gradient, i.e. each RTT interval.

We merged, processed and analyzed datasets using R to arrive at the following variables:

- *ertt* is the RTT estimated by Ertt, in milliseconds.
- $rtt_1 = ack_1 - tx_1$ is a reconstruction of the RTT, in microseconds, using the same packet pairing and clock as *ertt*, but without truncating the microsecond timestamp to milliseconds.
- $rtt_2 = ack_2 - tx_2$ is a reconstruction of the RTT, in microseconds, using the same packet pairing as *ertt* and rtt_1, but with a more accurate clock.
- spp is the reference RTT estimated by SPP, in microseconds, using packet traces. For fair comparison, we included the processing delay that SPP subtracts by default, refer to §4.1.1 ("RTT analysis").

We validated that the datasets *ertt*, rtt_1, and rtt_2 were correctly merged by comparing *ertt* to a reconstructed Ertt from tx_1 and ack_1 that used Algorithm 4.2, and arrived at identical results for all rows.

Results

Our first experiment runs indicated that there was a larger error in measurements than what could be explained by Bound 4.4. We found that the clock returned by `getmicrouptime()` does not progress evenly with real time, and thus breaks our accuracy assumption. Documentation states that:

> "The microuptime(), ... functions always query the timecounter to return the current time as precisely as possible. Whereas getmicrouptime(), ... functions are abstractions which return a less precise, but faster to obtain, time." [72]

The function `getmicrouptime()` ensures a call to `microuptime()` internally, but it returns a cached value on successive calls. Our next experiment runs logged results from both aforementioned clocks. We have omitted data from the first experiment for brevity, and only present results form our last experiment runs.

Table 4.1 describes the distribution of four different RTT signals, where *spp* is the most accurate, and *ertt* is used by CDG in FreeBSD.

RTT/delay	Var.	Mean	Min	Q1	Med.	Q3	Max
2×10 ms propagation delay							
ertt	274.606	42.625	21	27	41	54	105
rtt_1	272.176	41.625	20.288	26.610	40.011	52.474	104.215
rtt_2	272.069	41.512	20.280	26.573	39.982	52.087	104.178
spp	272.075	41.500	20.266	26.562	39.971	52.075	104.167
2×20 ms propagation delay							
ertt	108.211	51.363	41	43	48	NA	112
rtt_1	108.128	50.456	40.260	41.762	47.211	NA	110.315
rtt_2	107.951	50.336	40.232	41.684	47.175	NA	110.277
spp	107.957	50.323	40.220	41.672	47.163	55.662	110.266
2×35 ms propagation delay							
ertt	26.334	75.596	71	72	73	NA	165
rtt_1	26.510	74.651	70.284	71.613	72.129	NA	164.018
rtt_2	26.416	74.536	70.262	71.585	71.774	NA	163.579
spp	26.421	74.523	70.248	71.573	71.763	76.273	163.568

Table 4.1: RTT distributions in millisecond units, where rtt_1 uses the same clock as *ertt*, rtt_2 uses a more precise clock, and *spp* is estimated by SPP.

Table 4.2 compares all four signals to quantify their difference. The measured noise by $ertt - rtt_1$ matches the lower of Bound 4.4, but falls short of the upper by 8 µs. Possible explanations could be that we did not have enough samples to see the full range of possible values, that the clock granularity is coarser than 1 µs, or that some other mechanism in the sender, router or receiver runs at a rate less than 1 MHz.

The measured noise by $ertt - rtt_2$ quantifies the possible gain that the Enhanced RTT module could achieve by changing clocks. Separating noise on the acknowledgement path from the transmission path, $ack_2 - ack_1$ and $tx_2 - tx_1$ respectively, suggest that this noise is mainly induced on transmission. An observation from looking at the raw data is that a burst of segments transmitted during initial slow start receives identical tx_1 timestamps, even though tx_2 timestamps change. A few data with negative noise are observed; these are attributable to the time passing between calls to `microuptime()` and `getmicrouptime()`.

Variables	Var.	Mean	Min	Q1	Med.	Q3	Max
$ertt - rtt_1$	0.183	0.993	0.001	0.635	1.001	1.290	1.991
$ertt - rtt_2$	0.234	1.104	0.015	0.718	1.130	1.423	2.948
$ertt - spp$	0.234	1.116	0.027	0.730	1.141	1.434	4.536
$rtt_1 - rtt_2$	0.039	0.110	−0.001	0.028	0.037	0.044	1.01
$ack_2 - ack_1$	0	0	−0.007	0	0	0	0
$tx_2 - tx_1$	0.039	0.110	−0.001	0.028	0.037	0.043	1.01

Table 4.2: Sources of noise in milliseconds units. All propagation delays.

Prop. delay	Var.	Mean	Min	Q1	Med.	Q3	Max
Min-filtered measurement $rtt_{\min ertt} - rtt_{\min rtt_1}$							
$2 \times 10\,\text{ms}$	0.124	0.743	0.000	0.453	0.730	0.966	1.997
$2 \times 20\,\text{ms}$	0.108	0.769	0.000	0.553	0.755	0.904	1.992
$2 \times 35\,\text{ms}$	0.069	0.824	0.000	0.745	0.788	0.844	1.997
Max-filtered measurement $rtt_{\max ertt} - rtt_{\max rtt_1}$							
$2 \times 10\,\text{ms}$	0.128	1.315	0.007	1.058	1.298	1.578	1.995
$2 \times 20\,\text{ms}$	0.133	1.358	0.002	1.035	1.291	1.564	1.996
$2 \times 35\,\text{ms}$	0.150	1.375	0.003	1.053	1.296	1.583	1.996

Table 4.3: Noise induced by truncating timestamps, after min-/max-filtering.

We can also show and quantify the noise that Ertt's truncation of timestamps induces on the delay gradient signal. Gradients are computed from two measurements of RTT. Ertt's error bound for a single gradient is thus twice that of a single RTT measurement (Bound 4.4):

$$-2000\,\mu\text{s} < g_{rttm} - g_{rtt} < 2000\,\mu\text{s}. \tag{4.5}$$

Table 4.3 supports the claim in §2.6.1 that filtering reduces noise in the RTT signal, and suggests that a gradient signal could be less sensitive to noise than anticipated by doubling the error of a single measurement. A possible explanation is that the filtering step systematically skews measurements into extremums of the error range which has less oscillation. This skewness does not affect the outcome when sufficiently masked by the gradient operation.

Prop. delay	Var.	Mean	Min	Q1	Median	Q3	Max
Unsmoothed gradient of minimum $g_{\min ertt} - g_{\min rtt_1}$							
$2 \times 10\,\text{ms}$	0.241	-8.1×10^{-5}	−1.930	−0.372	−0.013	0.346	1.914
$2 \times 20\,\text{ms}$	0.213	-1.2×10^{-4}	−1.943	−0.253	−0.005	0.215	1.878
$2 \times 35\,\text{ms}$	0.130	-1.2×10^{-4}	−1.839	−0.057	−0.002	0.042	1.848
Unsmoothed gradient of maximum $g_{\max ertt} - g_{\max rtt_1}$							
$2 \times 10\,\text{ms}$	0.251	1.2×10^{-6}	−1.938	−0.387	−0.003	0.384	1.929
$2 \times 20\,\text{ms}$	0.263	-1.7×10^{-6}	−1.913	−0.367	−0.005	0.352	1.870
$2 \times 35\,\text{ms}$	0.291	3.3×10^{-5}	−1.869	−0.332	−0.007	0.338	1.907
Smoothed gradient of minimum $\hat{g}_{\min ertt} - \hat{g}_{\min rtt_1}$							
$2 \times 10\,\text{ms}$	0.004	-1.4×10^{-4}	−0.234	−0.043	0.000625	0.044	0.233
$2 \times 20\,\text{ms}$	0.003	-1.8×10^{-4}	−0.236	−0.036	0.000250	0.036	0.235
$2 \times 35\,\text{ms}$	0.002	-1.8×10^{-4}	−0.234	−0.019	0.000000	0.019	0.228
Smoothed gradient of maximum $\hat{g}_{\max ertt} - \hat{g}_{\max rtt_1}$							
$2 \times 10\,\text{ms}$	0.004	2.0×10^{-4}	−0.230	−0.042	0.000000	0.045	0.230
$2 \times 20\,\text{ms}$	0.004	1.4×10^{-4}	−0.232	−0.041	−0.000250	0.042	0.232
$2 \times 35\,\text{ms}$	0.004	3.1×10^{-4}	−0.234	−0.041	−0.000375	0.042	0.231

Table 4.4: Noise induced by truncating timestamps, after filtering step and gradient operation. Moving average uses right alignment and 8 gradients.

Table 4.4 show smoothed gradients outperforming unsmoothed gradients in terms of error range and variance. The moving average filter employed by CDG is "optimal for reducing random noise while retaining a sharp step response." [64] Random oscillations between each data point input to the filter can have the effect of canceling each other out, and outliers not part of the trend are reduced to a fraction of their value (1/8 using CDG's default window of 8 gradients).

Summary

We have analytically shown and experimentally quantified two sources of noise in measurements that are produced by the Enhanced RTT module. We have further estimated the noise induced in the resulting delay gradient signal used by CDG. However, we have not drawn conclusions about how this noise

affects the probability of delay gradient backoff. In the experiments used to quantify this noise, CDG would respond to a gradient signal constructed from *ertt* (the least accurate signal, i.e., unmodified CDG). The mean value of gradient noise, smoothed or unsmoothed, may appear to suggest that the backoff probability is not significantly affected. However, we are unable to draw inferences about backoff probabilities from the data collected so far, due to the fact that the backoff probability for a given flow by itself is dependent on the gradient signal, previous backoffs, and the other flows that are competing for capacity. Nonetheless, we suggest that future work could improve measurements made by the Ertt module based on our findings. This would serve as a general improvement of the delay signal in FreeBSD.

Precision and accuracy in Linux

We described how Linux performs RTT measurements in §3.3. Having established the precision and accuracy of Ertt's measurements, we can apply part of our analysis to Linux. We also suggest a possible avenue to improve this accuracy.

`skb_mstamp_get()` produces a socket timestamp ts (`struct skb_mstamp`):

$$\begin{aligned} ts_{jiffies} &\leftarrow jiffies, \\ ts_{us} &\leftarrow \lfloor \texttt{local\_clock}()/1000 \rfloor , \end{aligned} \tag{4.6}$$

where flooring follows from integer division. Jiffies are a kernel-specific time unit based on kernel tick rate, giving $1 - 10\,\text{ms}$ resolution for tick rates of $1000 - 100\,\text{Hz}$, respectfully. Jiffies are used as fallback when integer wrapping of us is detected between two timestamps. `local_clock()` returns the architecture-specific scheduling clock in nanoseconds. It follows that the resulting RTT measurements have a precision in the range of microseconds.

We establish a lower and upper bound for accuracy by trivial adaptation of the approach and assumptions in Bound 4.4:

$$0\,\mu\text{s} < rttm - rtt < 2\,\mu\text{s}. \tag{4.7}$$

We note that a possible improvement in accuracy can be achieved by rounding half-way cases upwards:

$$ts_{us} = \lfloor (\texttt{local\_clock}() + 500)/1000 \rfloor .$$

However, we have not evaluated whether this change is valuable and have not submitted a patch for it, e.g., it would depend on the accuracy of `local_clock()`, and whether this small increase in accuracy justifies the extra arithmetic operation per packet.

Chapter 5

Conclusions and future work

This chapter concludes our thesis on implementing the CAIA Delay-Gradient (CDG) congestion control in the Linux kernel. We discuss the findings of this thesis and present future work for CDG and congestion controls in general.

5.1 Conclusion

When evaluating a congestion control there are many considerations, including how fairly it can compete with instances of itself or other congestion controls, and how its performance varies in a vast range of diverse environments [25, 24, 35]. We have performed an experimental evaluation of our CDG implementation to evaluate its ability to share capacity with instances of itself, where our Linux implementation showed improved total throughput compared with the FreeBSD implementation for the grater share of experiment parameters. A limitation of our experimental evaluation is that we have not evaluated how it behaves in competition with other congestion controls or in highly multiplexed environments. However, our results indicate that our Linux implementation can compete effectively under our experiment parameters, and that it can operate more effectively than the FreeBSD implementation in terms of obtain throughput when it is not competing.

During our work to implement CDG in Linux, we encountered several technical challenges that led to results and contributions on related topics:

- We identified areas of improvement to Linux' RTT measurements for congestion control, namely the absence of RTT measurements from SACK, and a special case that produced an erroneous RTT measurement. We proposed five patches to address these areas to the Linux networking development community, and all patches were accepted as contributions to the Linux kernel.

- We have analytically shown and experimentally quantified two sources of noise in measurements that are produced by the Enhanced RTT module. We have further estimated the noise induced in the resulting delay gradient signal used by CDG.

The fundamental mechanisms that congestion controls rely on for operation appears to be implemented considerably different in Linux and FreeBSD operating systems, and our contributions to the Linux kernel have reduced that gap to some extent [32].

We made several adjustments or improvements to the design of CDG for our Linux implementation:

- We have improved the precision and accuracy of CDG's backoff function by developing a simple integer algorithm for computing the negative exponential backoff. This is a novel algorithm that may be reused for other purposes, e.g., in kernel code or embedded devices.

- We have enhanced CDG to perform shadow window validation.

- We have enhanced CDG to be responsive to ECN signals.

- We have integrated the use of Proportional Rate Reduction (PRR) in CDG.

- We added a toggle to enable or disable the shadow window that possibly increases the application of CDG as a Less-Than-Best-Effort congestion control. This was a result of a discussion with Dr. David Hayes.

- We made a small change to the loss tolerance heuristic's design that reduces the impact of an erroneous inference.

- We made a decision to disable the loss tolerance heuristic by default due to concerns about its safety outside closed environments, i.e., the Internet.

Finally, we concluded that, as is, our conservative precaution of disabling the loss tolerance heuristic by default makes CDG safe for experimental testing by a wider community, i.e., it is no less conservative than NewReno. An initial version of our CDG implementation was posted for review on the network development mailing list, and we received encouraging suggestions for improvements that need to be addressed before a final version can make its way into the Linux kernel.

5.2 Background traffic

As we described in §3.4.3, our Linux implementation allows both coexistence heuristics to be disabled by setting module parameters `use_shadow=0 ineffective_thresh=0`. It is not currently possible to similarly disable the shadow window in the FreeBSD implementation.

LEDBAT is a delay-based congestion control that is "designed for use by background bulk-transfer applications, ... and to yield in the presence of competing flows." [63]. However, as we described in §2.5.2, it has a potential weakness in that it incorporates its self-induced queueing delay into its base delay estimate. Through the use of delay gradients, CDG may be able to elude this pitfall of LEDBAT.

Previous experiments [5] have found that the existing FreeBSD implementation of CDG was a viable choice for background transport applications when a backoff beta parameter of 0.5 was used (default is 0.7), but they also found that CDG would retain some of its sending capability. We propose that disabling both of CDG's coexistence heuristics, the ineffectual backoff detection and the shadow window, may inhibit its ability to grow the congestion window at any significant capacity in the presence of loss-based competition. It may reduce its congestion window close to the bare minimum of 2 MSS using the default backoff beta of 0.7, and thus it may also achieve a better performance in the absence of competition from other flows. This could improve on CDG's suitability as a a congestion control for background traffic. CDG may thus be a serious contender to the applications of LEDBAT, but future research is necessary to draw conclusions.

5.3 Hybrid slow start

In our experiments, CDG did not always back off before slow start overshoot. We point out that the particular problem of slow start overshoot may be alleviated by use of the Hybrid Slow start algorithm [29, 21], and suggest implementing Hybrid Slow start in CDG as an area of future work. Decoupling the Hybrid Slow start algorithm [29] used by the CUBIC module is a possible first step.

5.4 Cubic's congestion avoidance

Like CUBIC, CDG's mechanisms aim to achieve fairness between flows with different RTTs. However, our results in §4.3 may indicate that flows with

high RTTs indeed suffer a relative unfairness since they are unable to regrow the window at the same pace at flows with low RTTs.

CDG inherits the AIMD and MIMD mechanisms of NewReno. A long history in the research community makes NewReno attractive as a base for experimental congestion control mechanisms. With Hybrid Slow start in place, it would be useful to evaluate whether incorporating CUBIC's congestion avoidance improves performance. A new congestion control hybrid of CDG and CUBIC could possibly be better than either algorithm alone.

5.5 One-way delay

Linux does not generally support one-way delay measurements at time of writing. An implementation could serve as an improvement for all delay-based congestion controls. In the case of using OWD with CDG, the algorithm does not need to be changed. The gradients of OWD will be comparable to gradients of RTT with the added benefit of avoiding delay fluctuations in the reverse path that are uncorrelated with congestion in the forward path.

An implementation may consider using the TCP timestamp option for this purpose – it conveys local timestamps from both ends of a TCP connection to the other party. As is, there are only weak requirements to the precision of a valid TCP timestamp.[1] The problem for OWD measurements is that a much finer granularity is required to compete with existing RTT measurements, and reliably extracting precise timing information from a TCP timestamp requires a change in protocol between sending and receiving TCPs. Scheffenegger et al. has proposed incrementally deployable methods to solve this problem, either through stating timestamp precision, or negotiating timestamp capabilities [61, 62].

Notice that, as is, the timestamp echo reply field (ECR) reflects the oldest received segment during delayed acknowledgement, loss or reordering. The proposal for timestamp negotiation opens for changing it to the latest received segment. This may mostly be useful for distinguishing original and retransmitted segments. It could be best for a Linux implementation to only rely on internal timestamps stored in the retransmission queue due to the concern that a receiver or middle-box in between may knowingly or unknowingly manipulate the ECR field. The sanity of timestamps originating from the receiver should be validated for the same reason. This can be

[1] At time of writing, FreeBSD uses `tcp_ts_getticks()` for TCP timestamps (see §4.4), and Linux uses *jiffies* (see §3.3). It is plausible to detect a receiver's timestamp granularity by heuristic, e.g., Linux' TCP Low Priority congestion control module does this (see `net/ipv4/tcp_lp.c` in kernel sources).

done by demanding one-way delays to be somewhere within the range of supplemental RTT timings, and possibly by use of a heuristic that validates newly computed one-way delays against previous computations, e.g., with an exponentially smoothed average.

5.6 Magic numbers

The backoff probability, and indirectly the trade-off between throughput and delay, is both influenced by the scaling parameter and the smoothing factor. Both are considered to be "magic numbers", i.e., chosen because they work under a certain scenario [4, 01:06–]. It would be beneficial to have more substantial recommendations on choice of parameters.

5.7 Hardware RTT measurements

Recent network adapter support hardware timestamping. This is a desirable property for delay-based congestion control to avoid processing delays such as offloading in the network adapter, scheduling interruptions by the operating system, and clock inaccuracies.

Currently, Linux uses the scheduling clock for retransmission timeouts. This clock is not necessarily accurate with real time. An implementation of hardware timestamps for congestion control would have to evaluate whether these can be used for RTO timers or not, e.g., to avoid spurious retransmits.

5.8 New tcpprobe functionality

Linux provides a general-purpose tcpprobe module for recording congestion control parameters. It works by installing a function hook on entry to `tcp_rcv_established()`. Similarly, Linux provides the means to attach kernel hooks on entry or return of any known function symbol or address.

It could be beneficial to extend tcpprobe so that it is able to record events and internal data that are tailored to specific congestion control modules. One approach would be to attach function hooks on the operations that these modules provide. Pointers to these operations (`struct tcp_congestion_ops`) are internal to `net/ipv4/tcp_cong.c` at time of writing, and acquiring them from a kernel module would demand a new function to export it. `tcp_ca_find()` can be adapted for this purpose. Care must be taken to avoid races with module unloading; it could be sufficient to increment the congestion control module's reference count while holding the `tcp_cong_list` spinlock.

5.9 ECN-based congestion control

DataCenter TCP (DCTCP) changes the sender to estimate the ratio of packets that experienced congestion to the packets successfully transferred. DCTCP has been shown to increase network efficiency for certain workloads in closed environments [1], but has severe drawbacks that makes it unfit in the general Internet [18]. As described in 2.5.3, it uses a proprietary handshake that can threaten future standardization efforts, and lacks reliability of receiver feedback. One experimental proposal that address these concerns is known as *More accurate ECN feedback in TCP* (AccECN) [17]. Experimental efforts to implement AccECN as proposed may be a possible first step to tackle the aforementioned issues.

Appendix A

Time series data from experiments

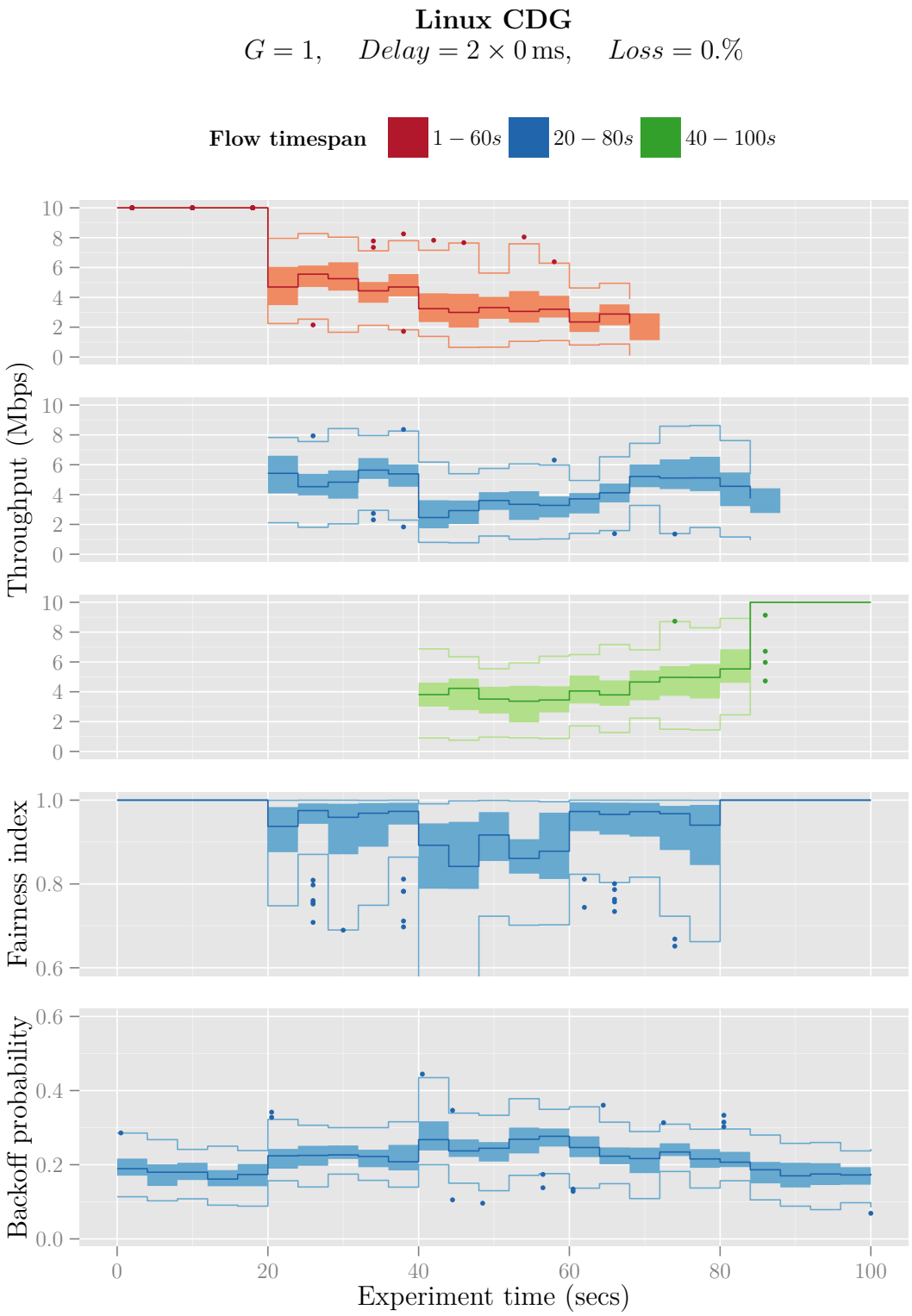


Figure A.1: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

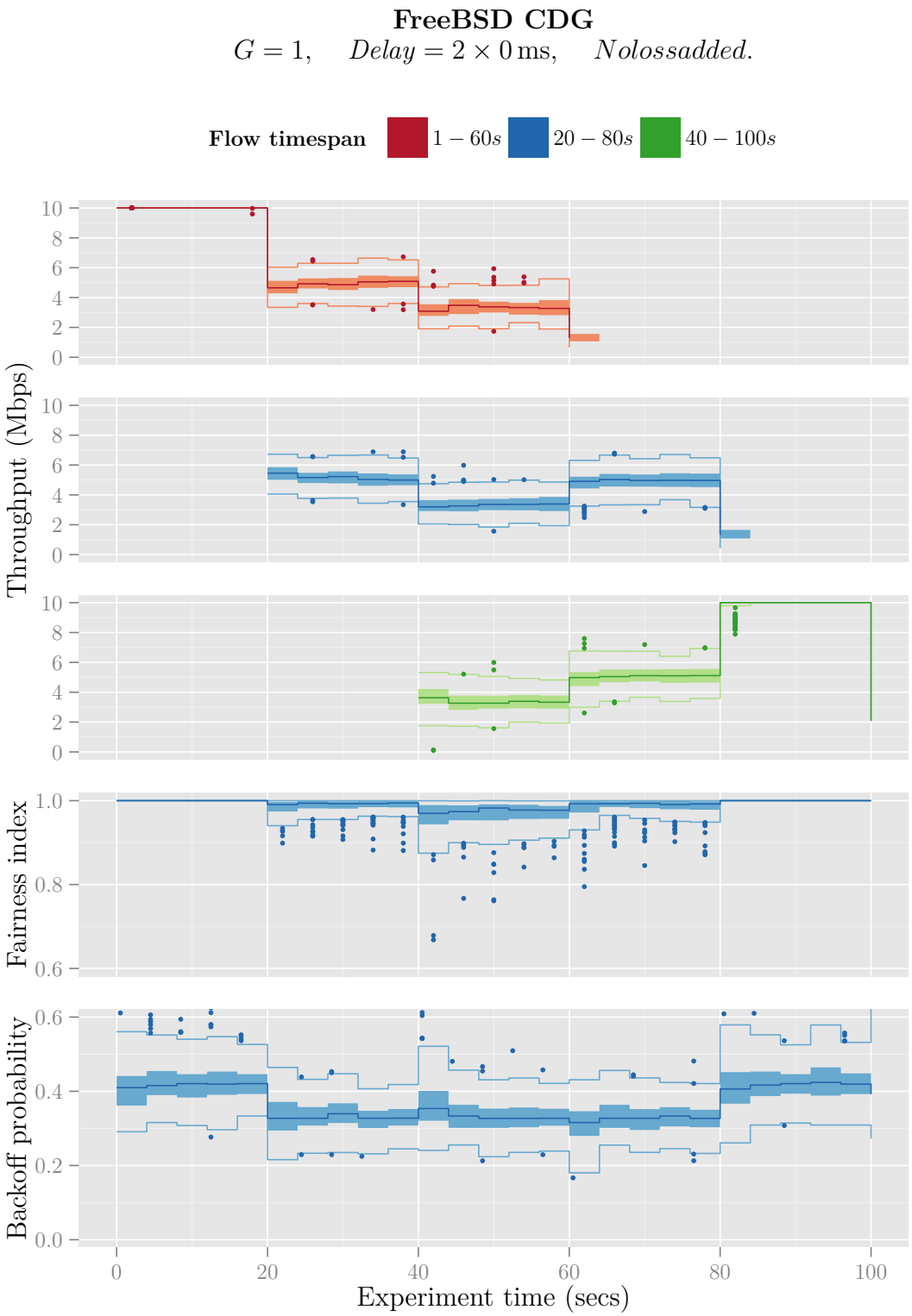


Figure A.2: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

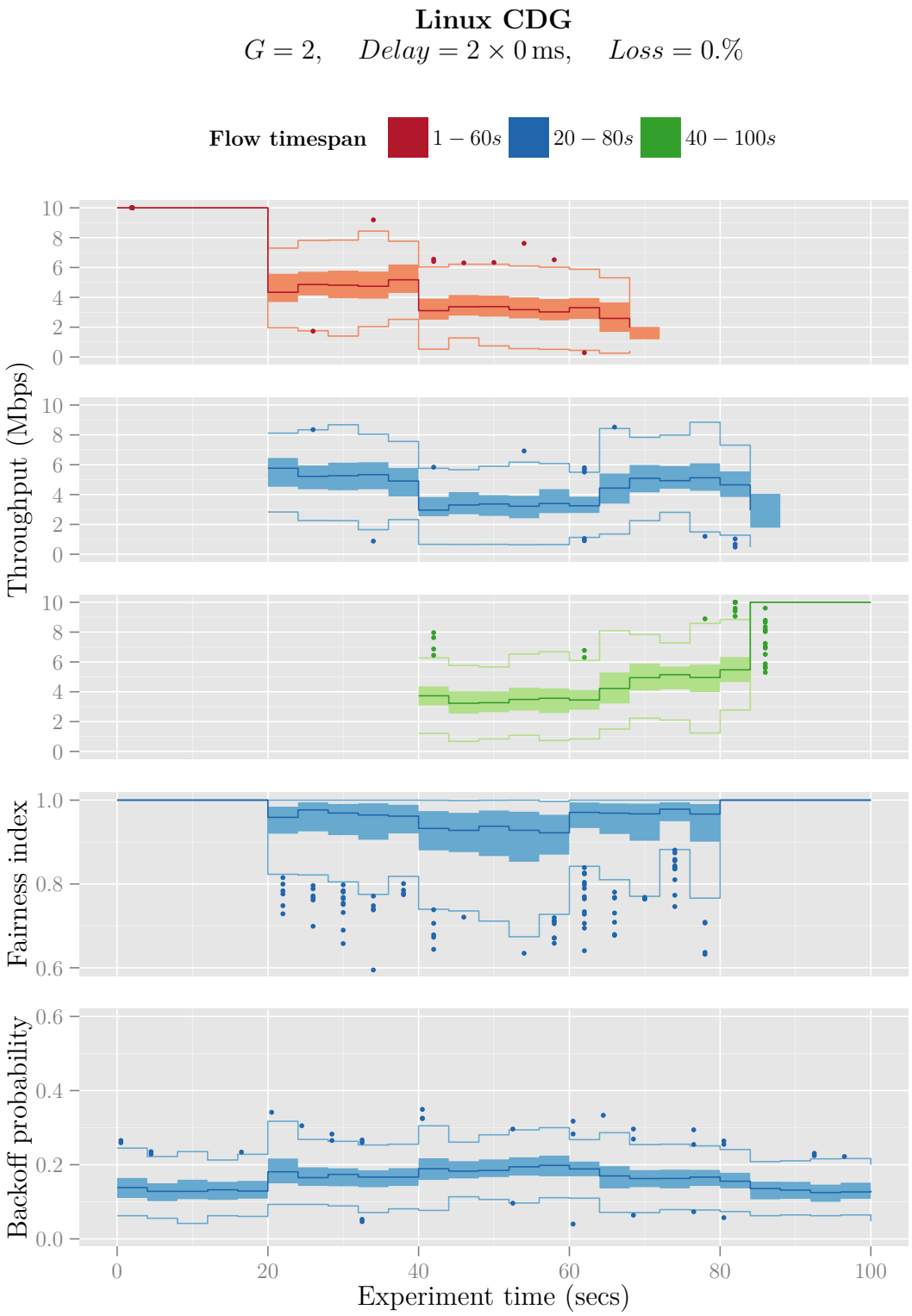


Figure A.3: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

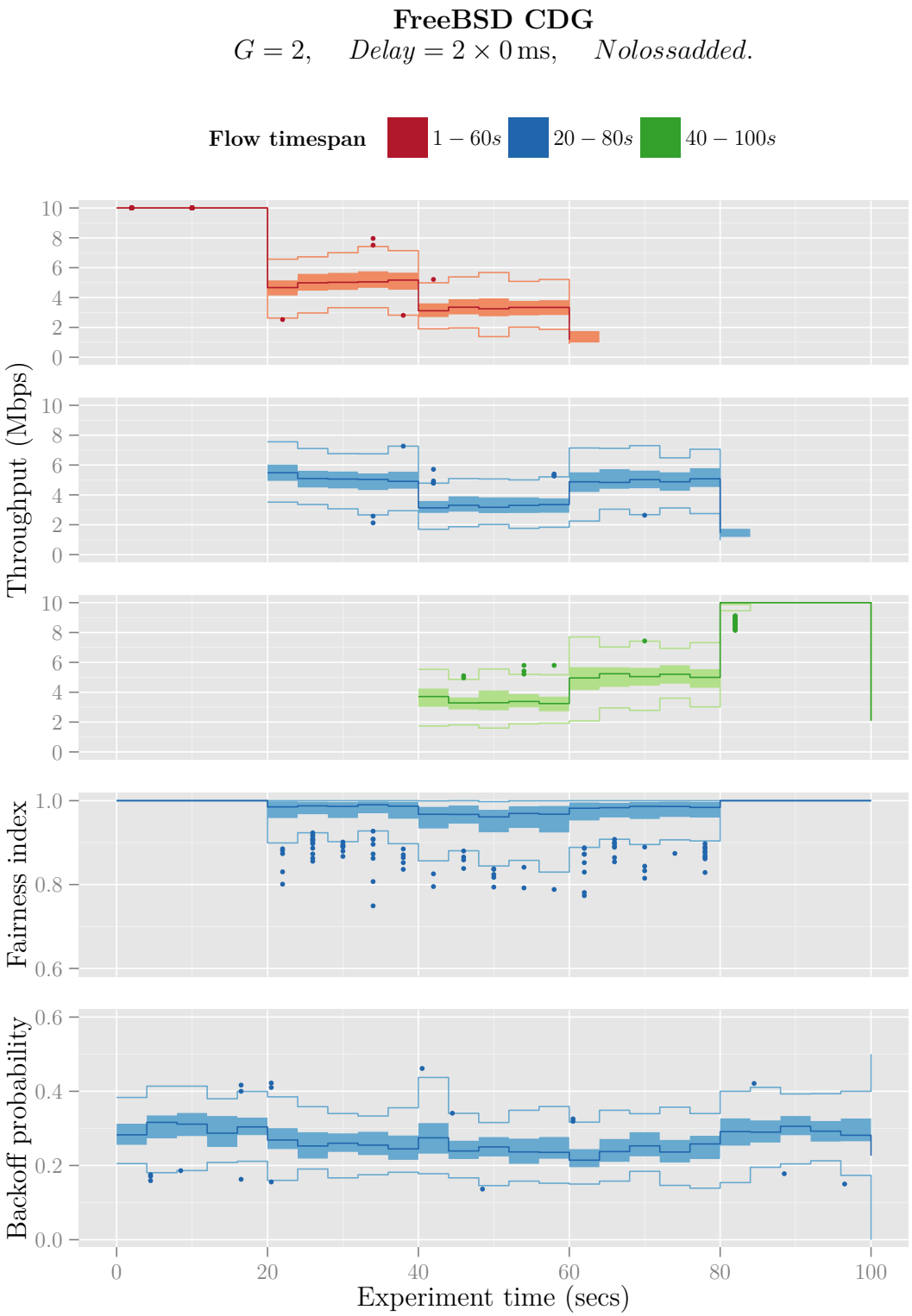


Figure A.4: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

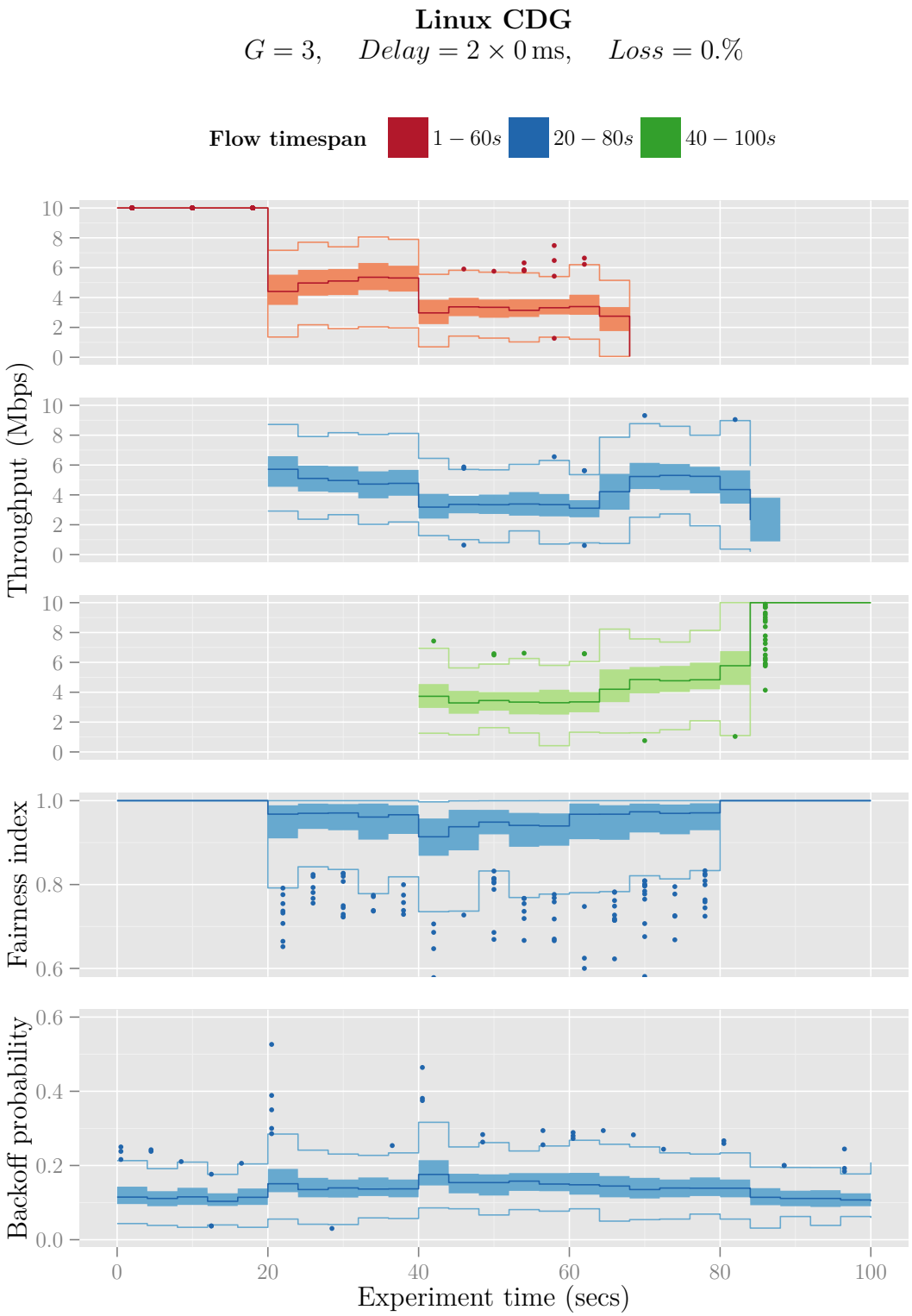


Figure A.5: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

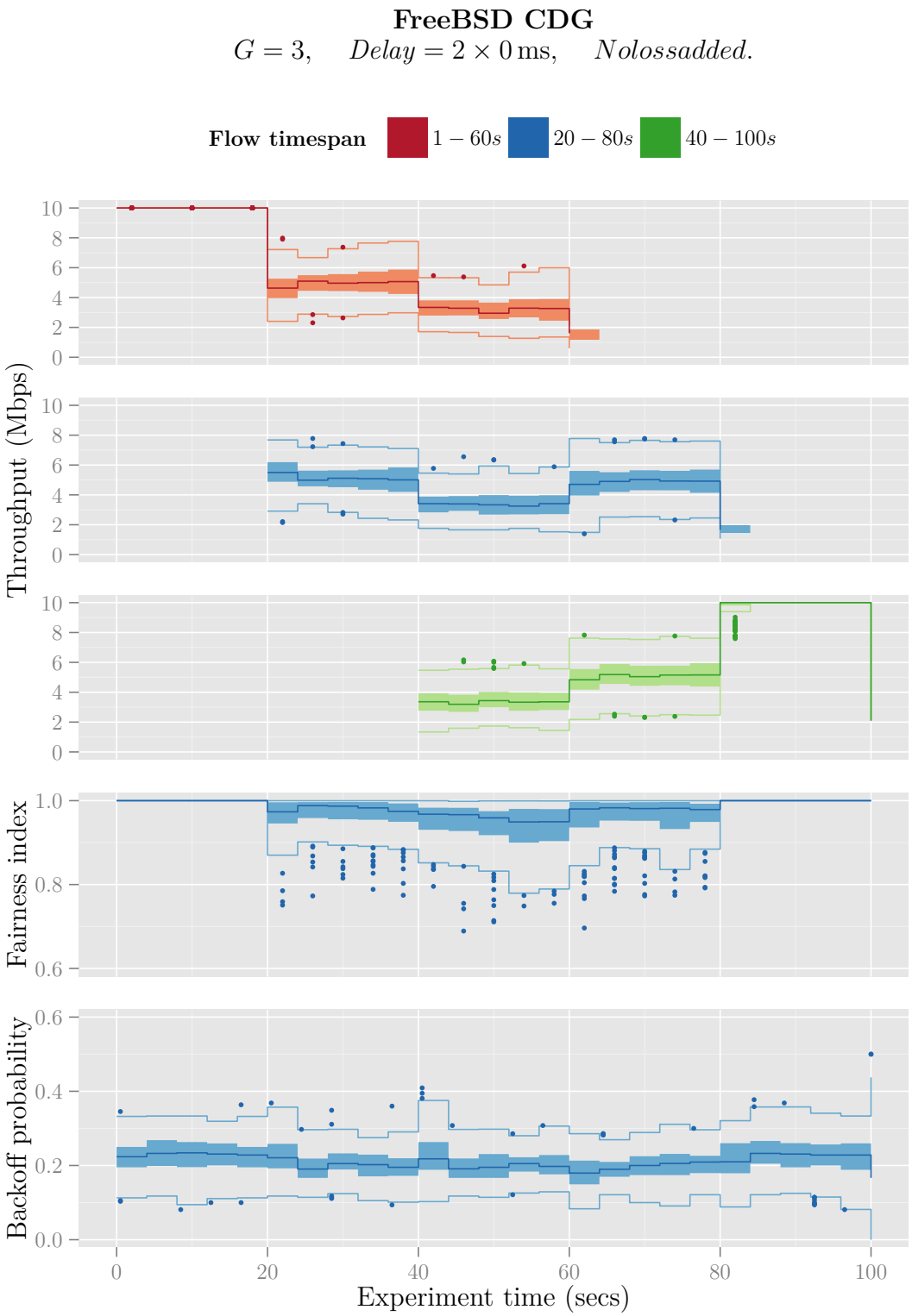


Figure A.6: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

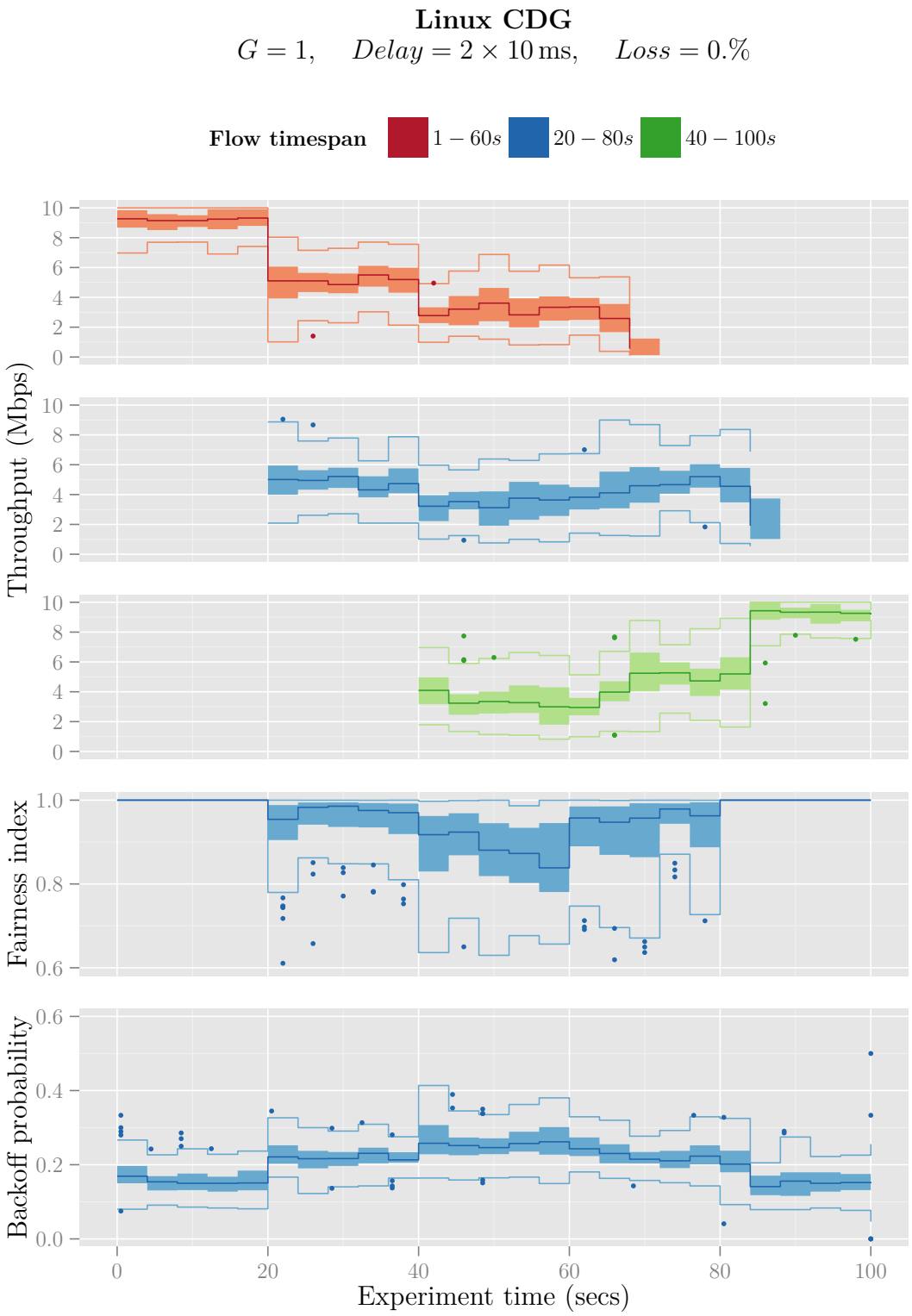


Figure A.7: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

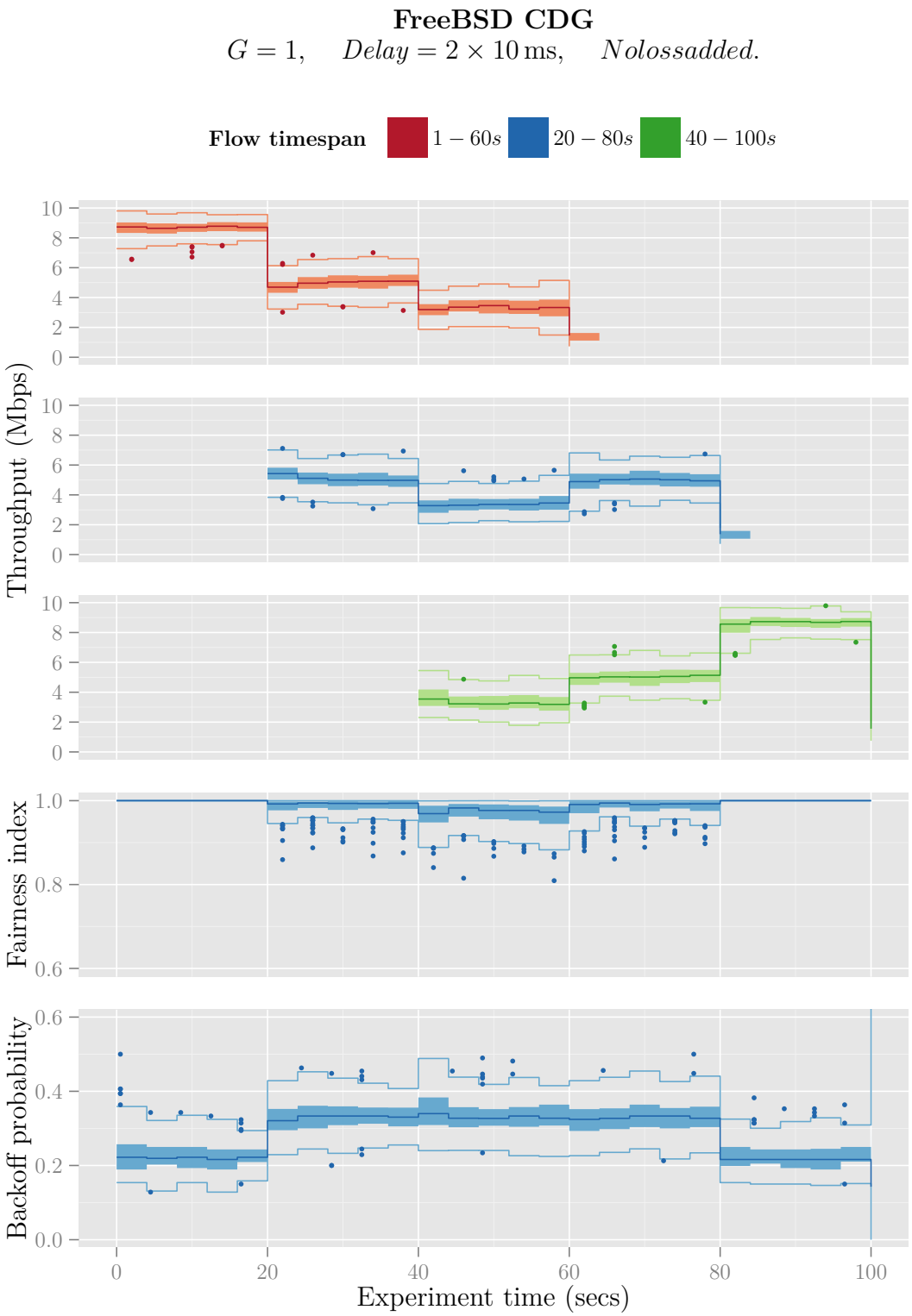


Figure A.8: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

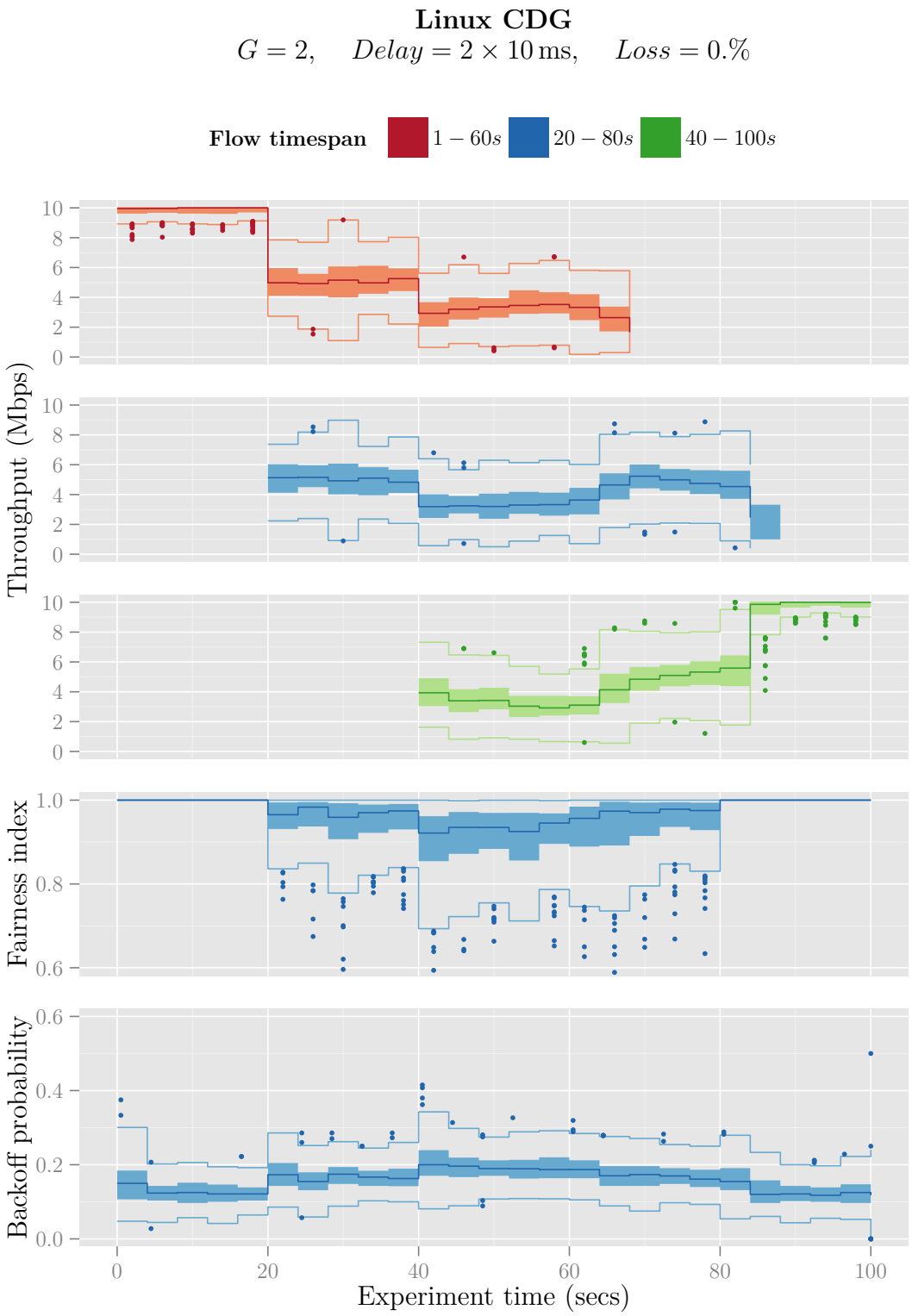


Figure A.9: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

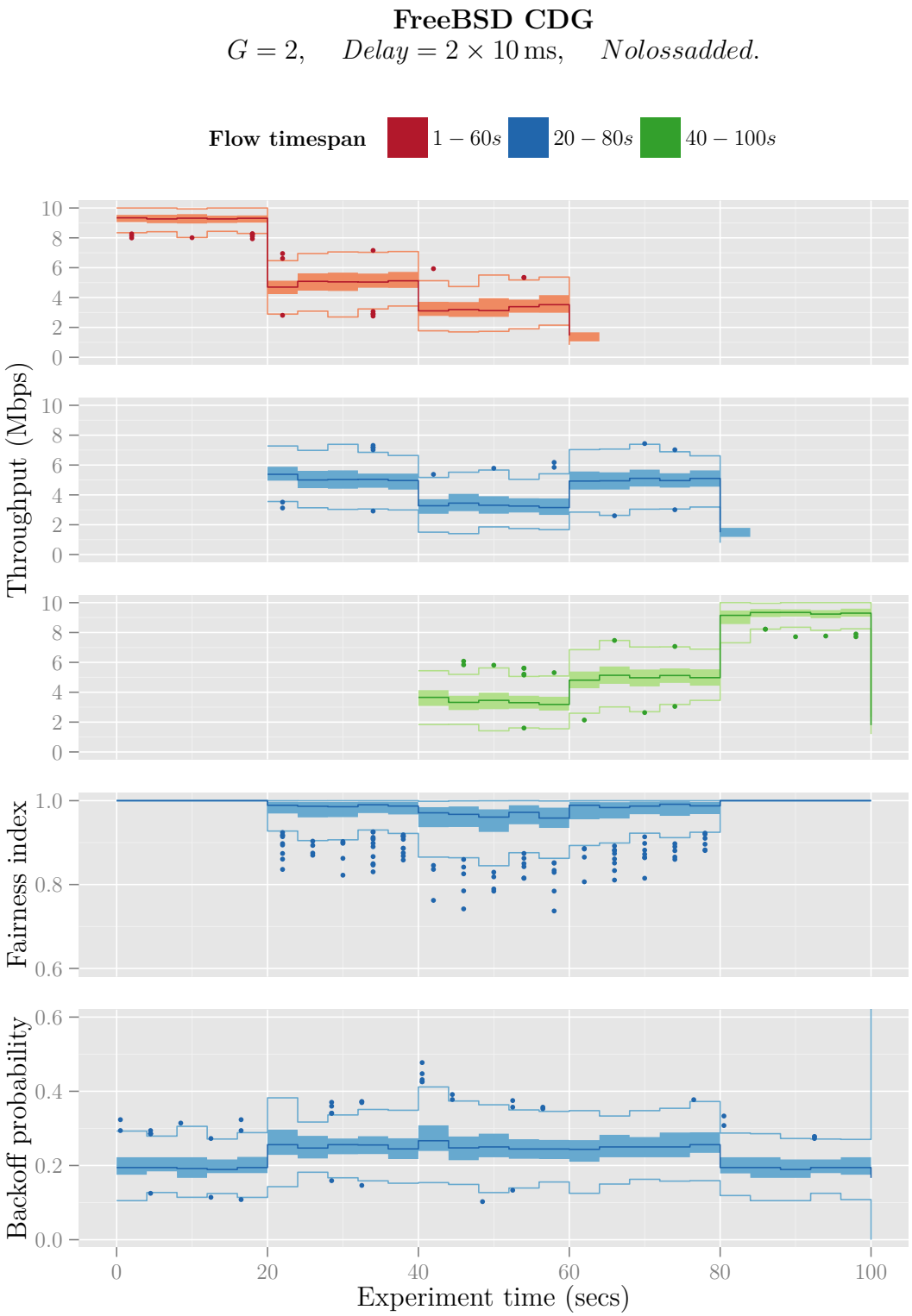


Figure A.10: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

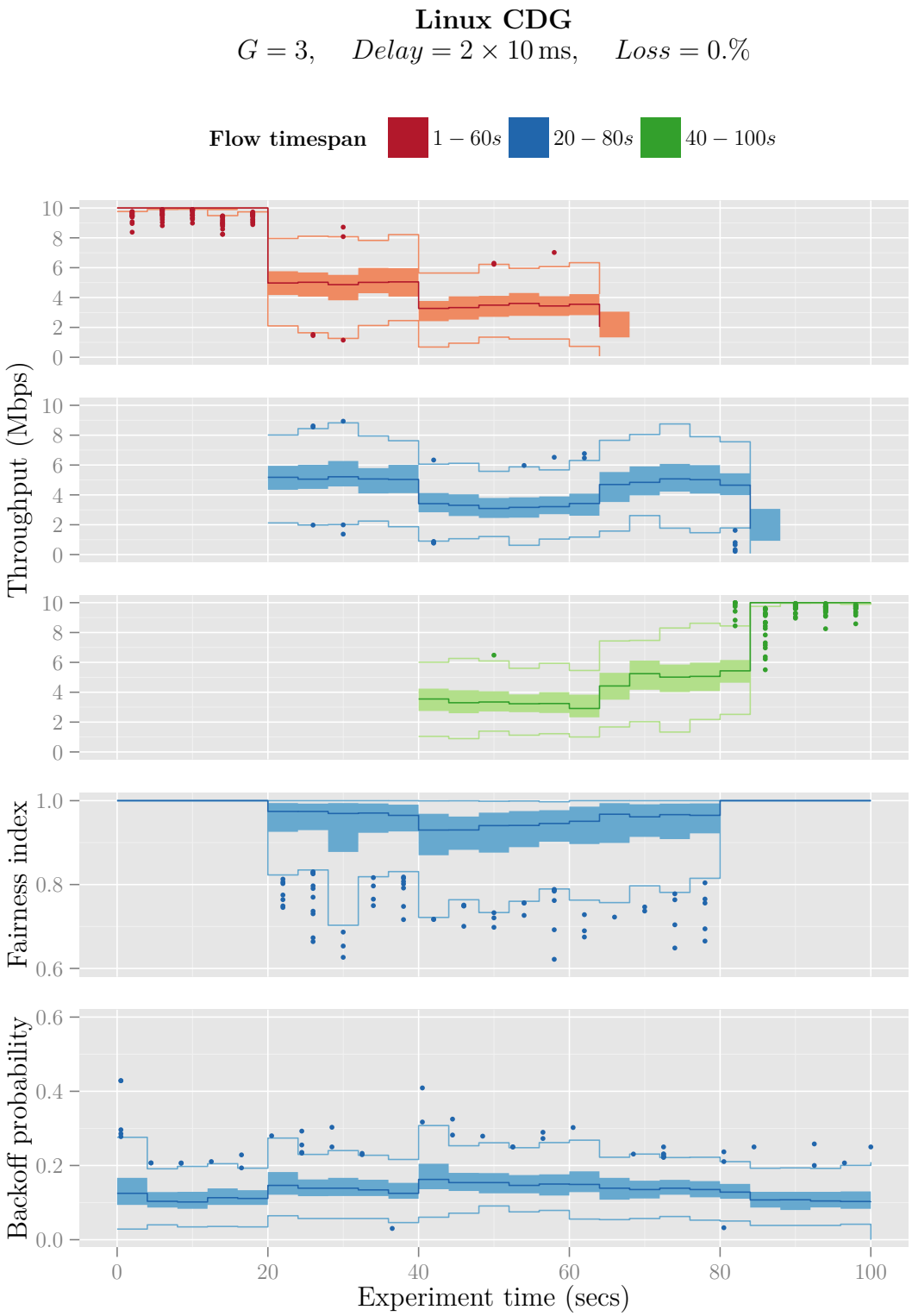


Figure A.11: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

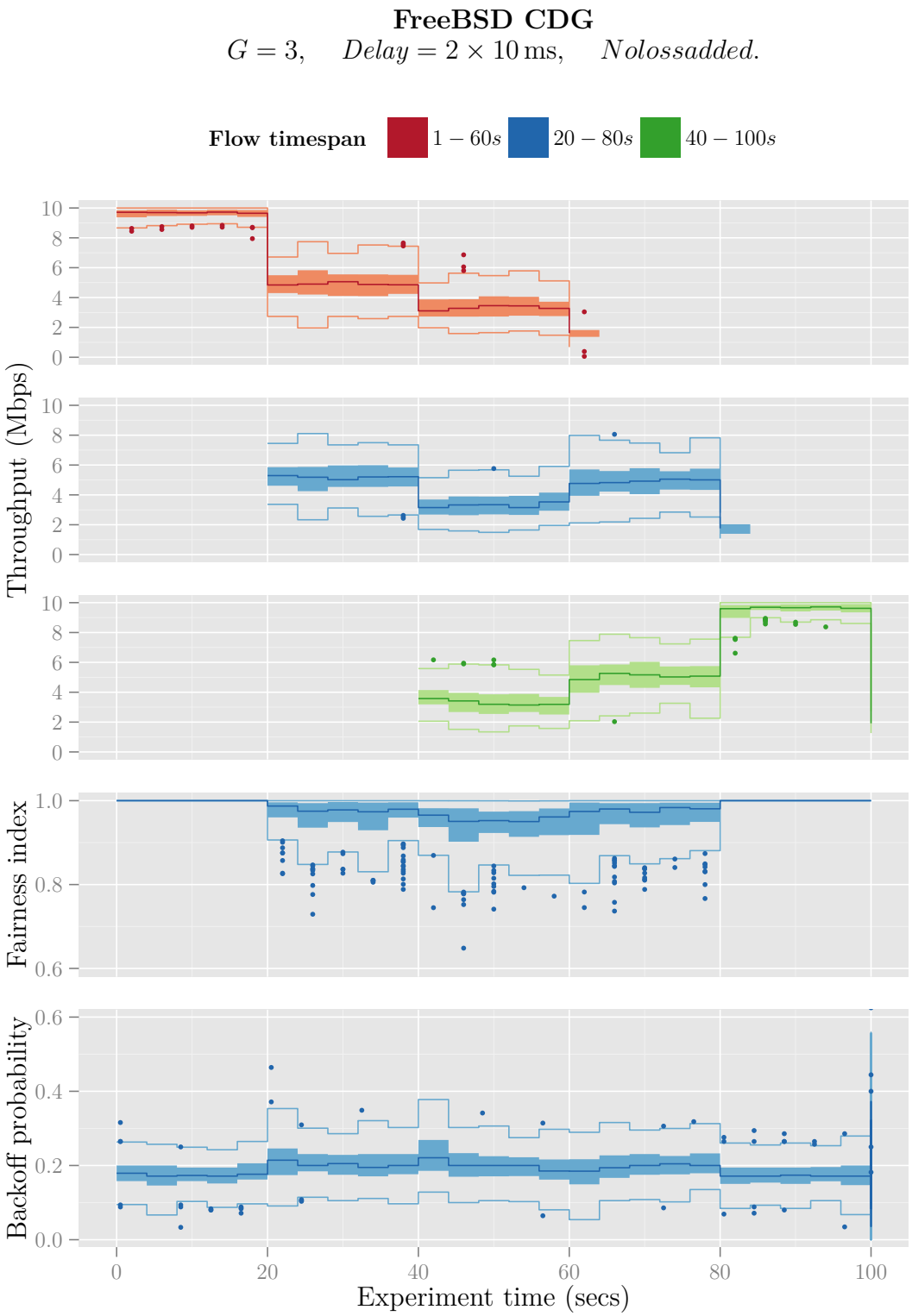


Figure A.12: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

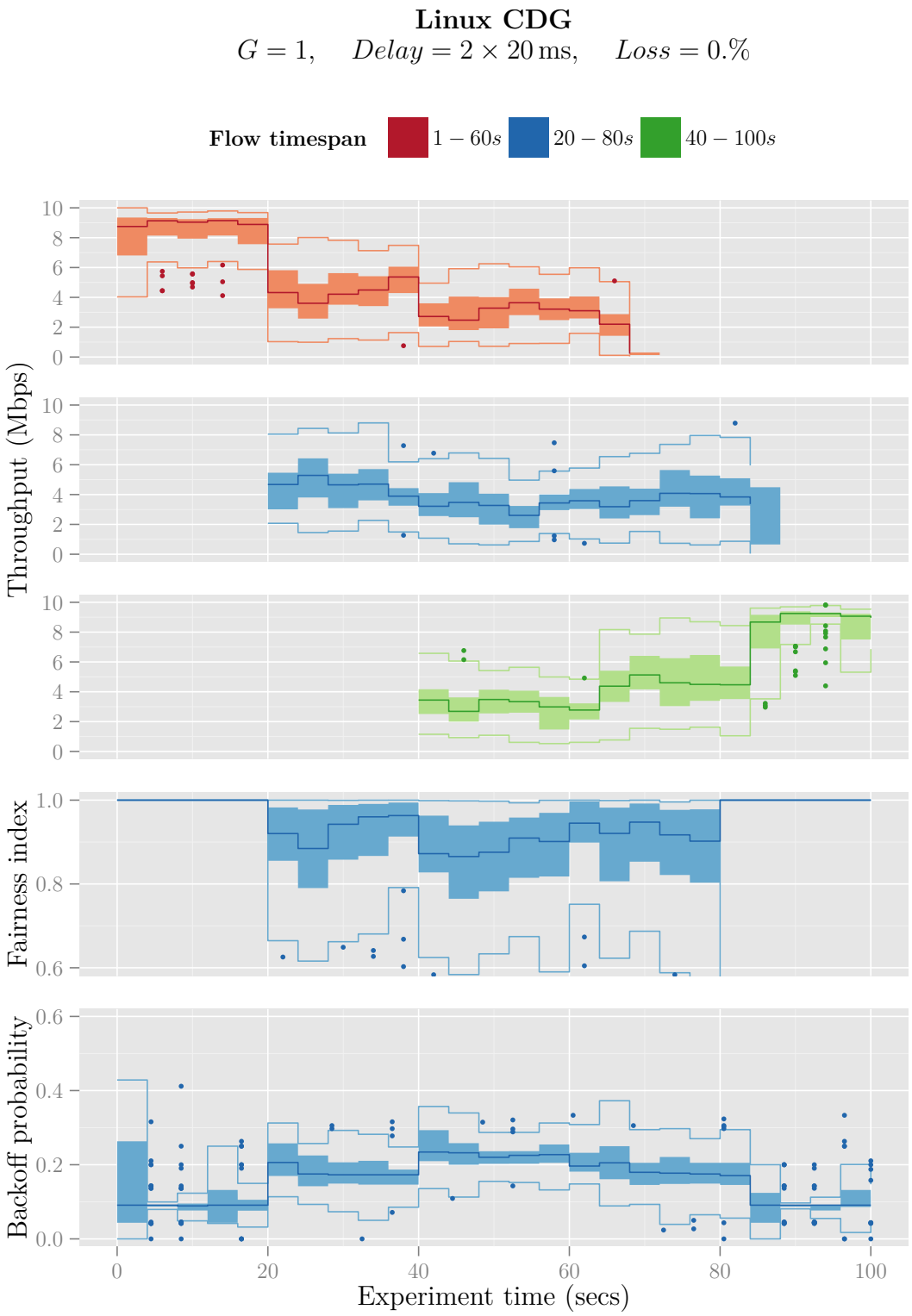


Figure A.13: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

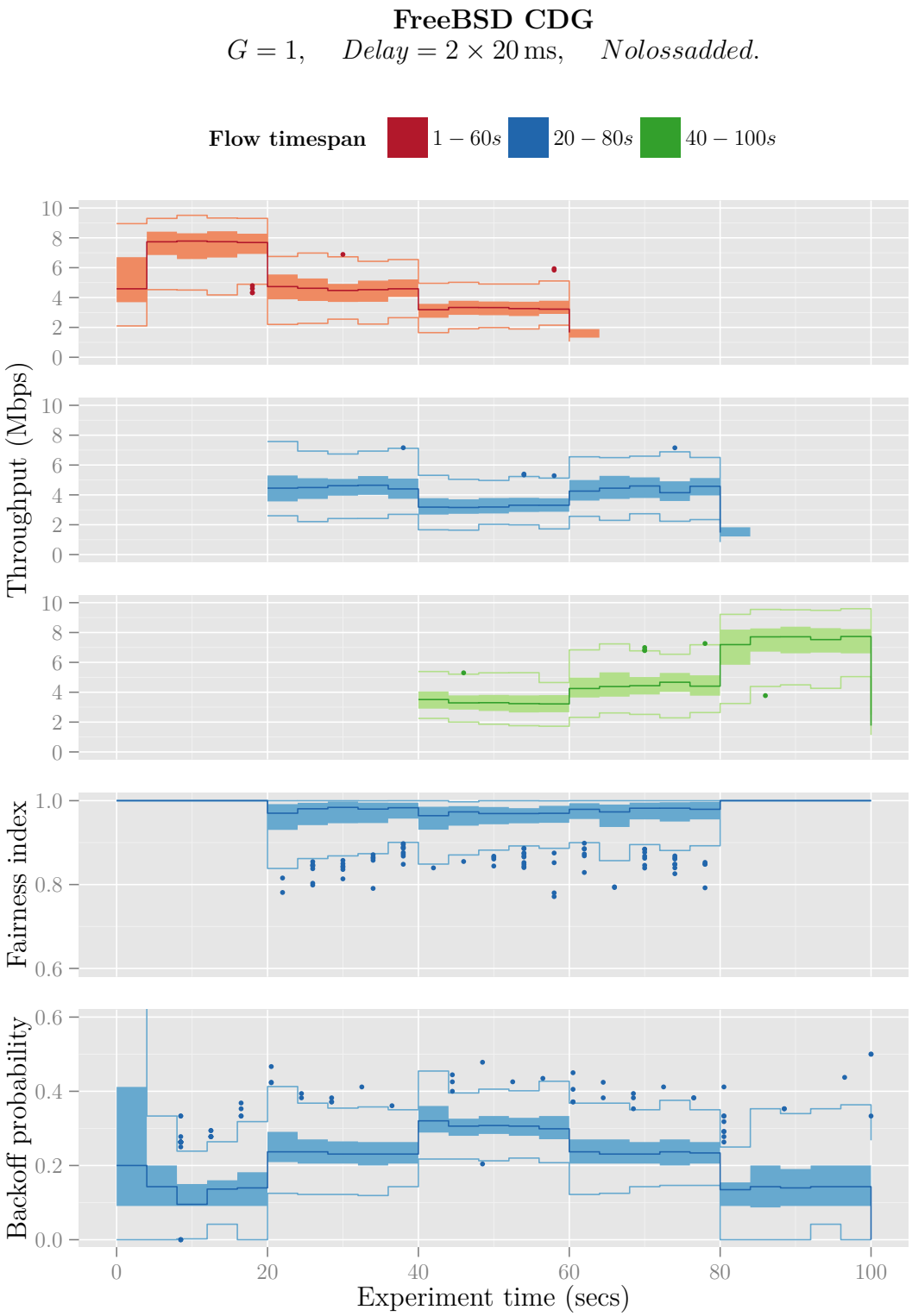


Figure A.14: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

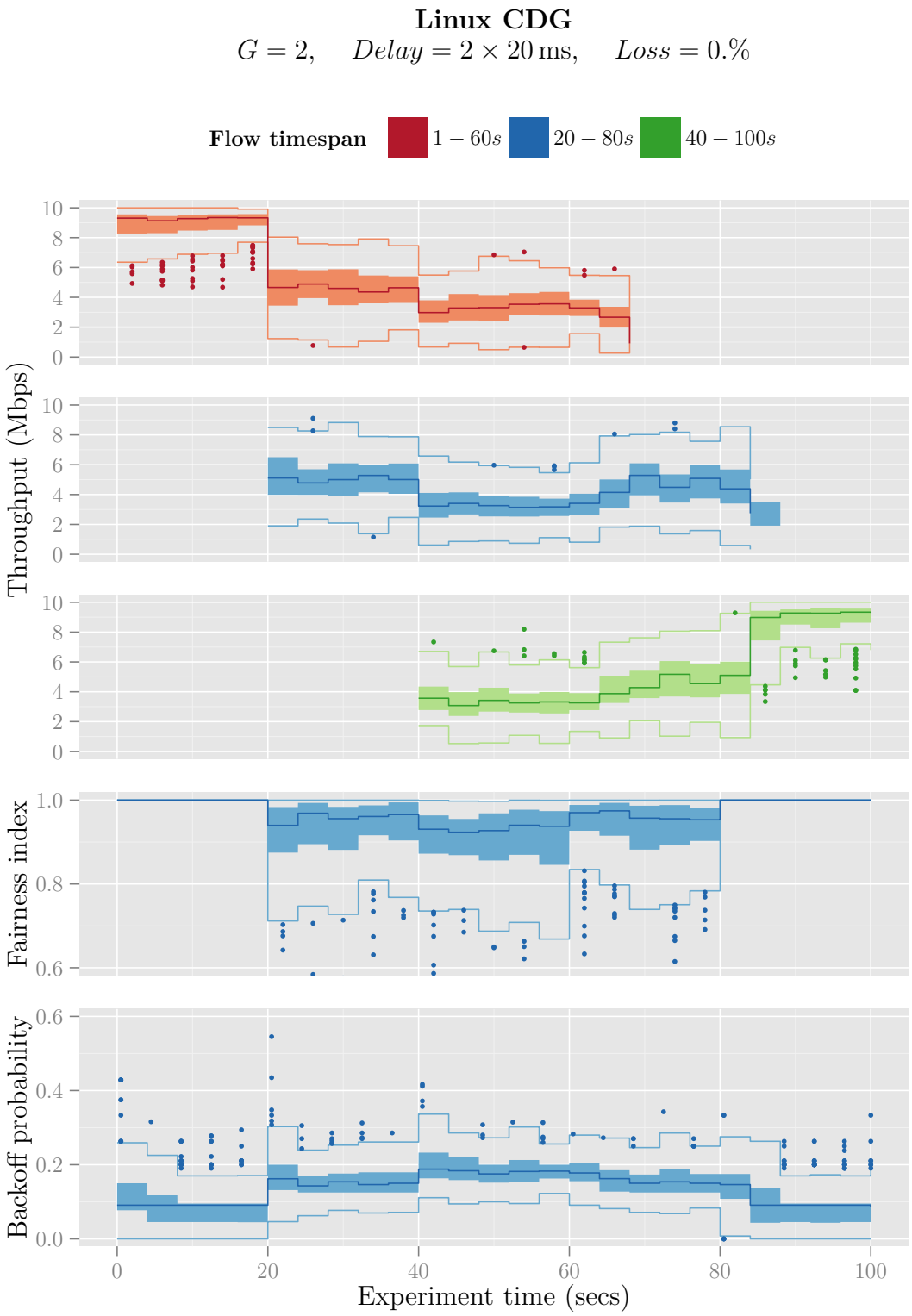


Figure A.15: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

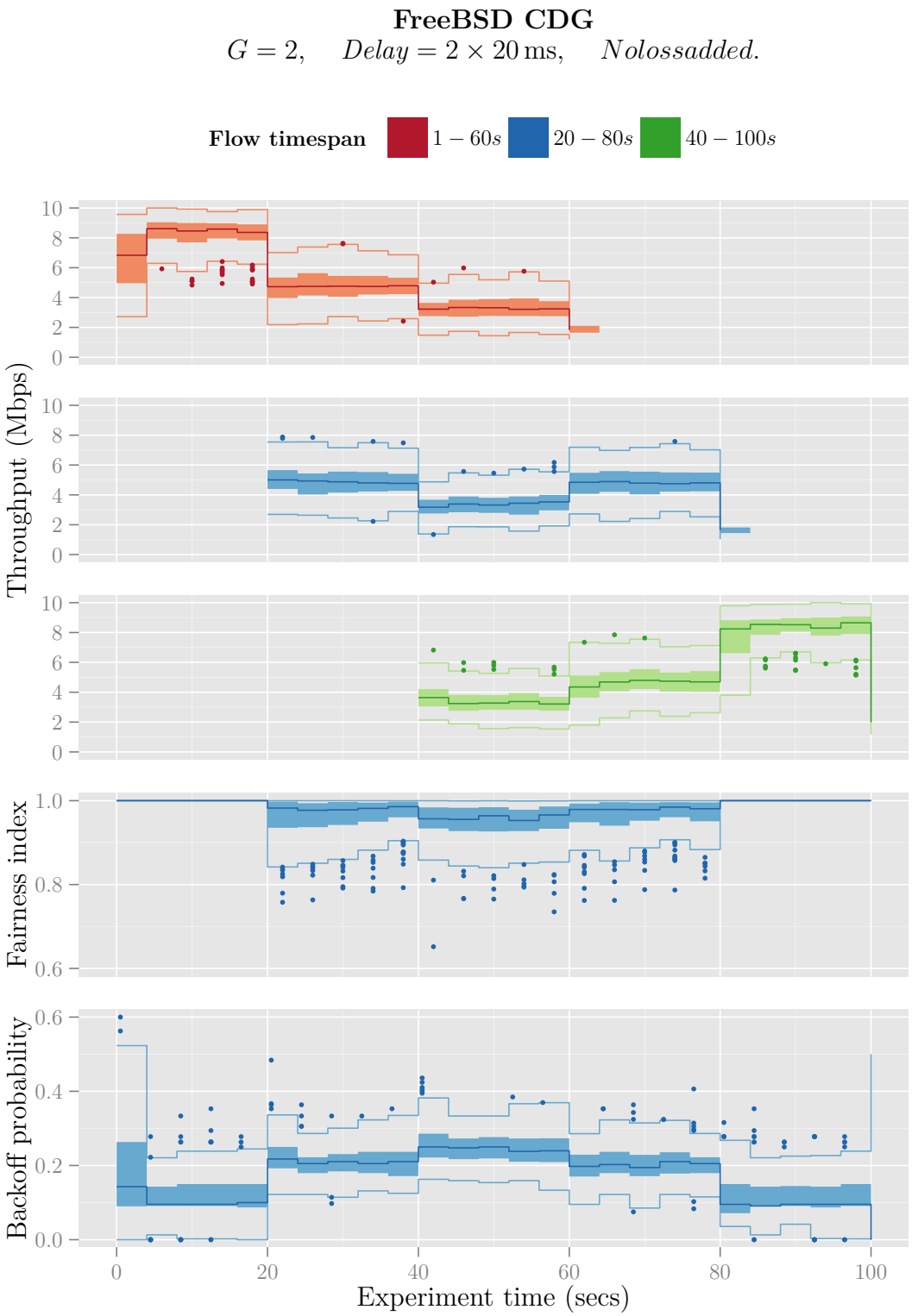


Figure A.16: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

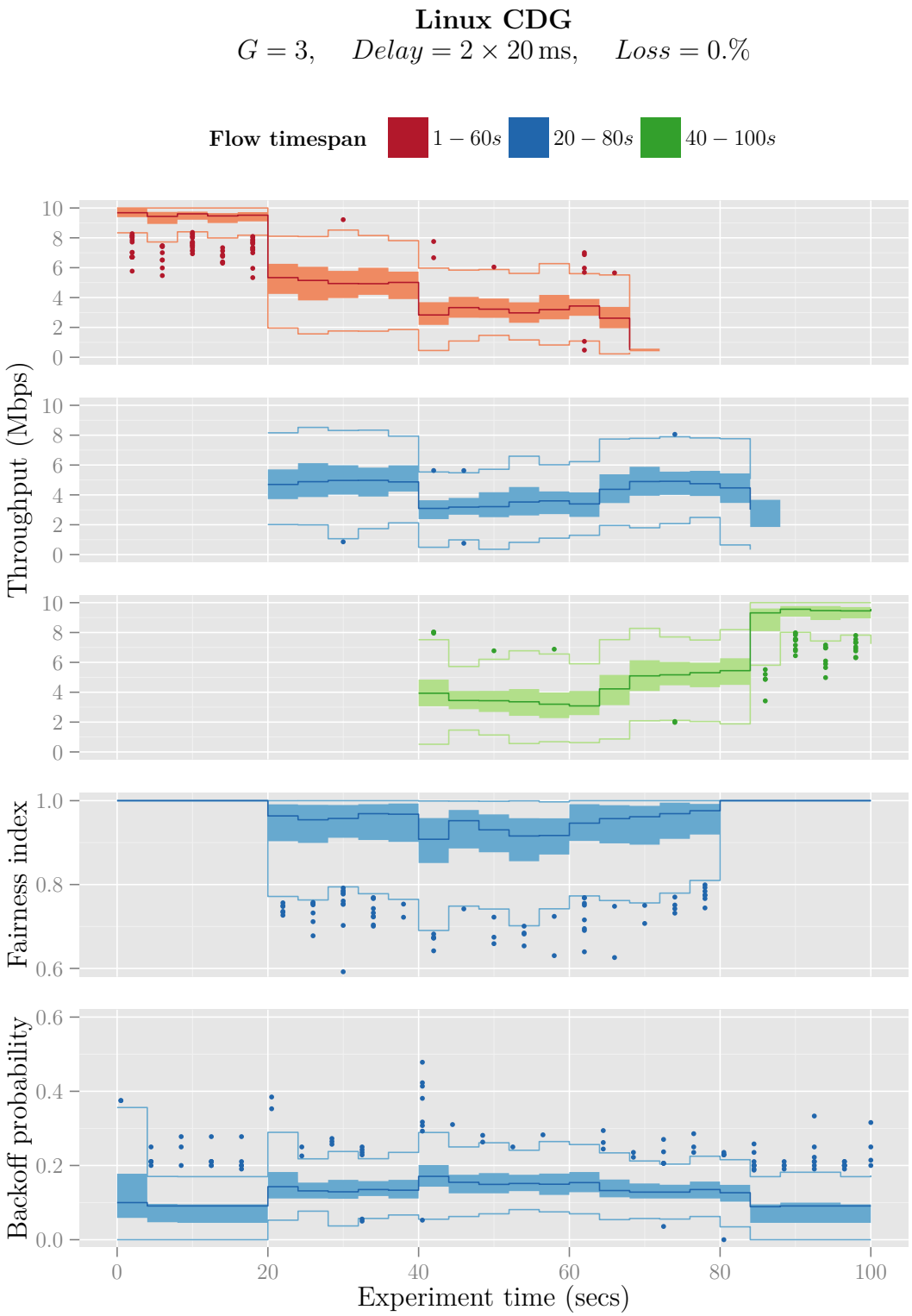


Figure A.17: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

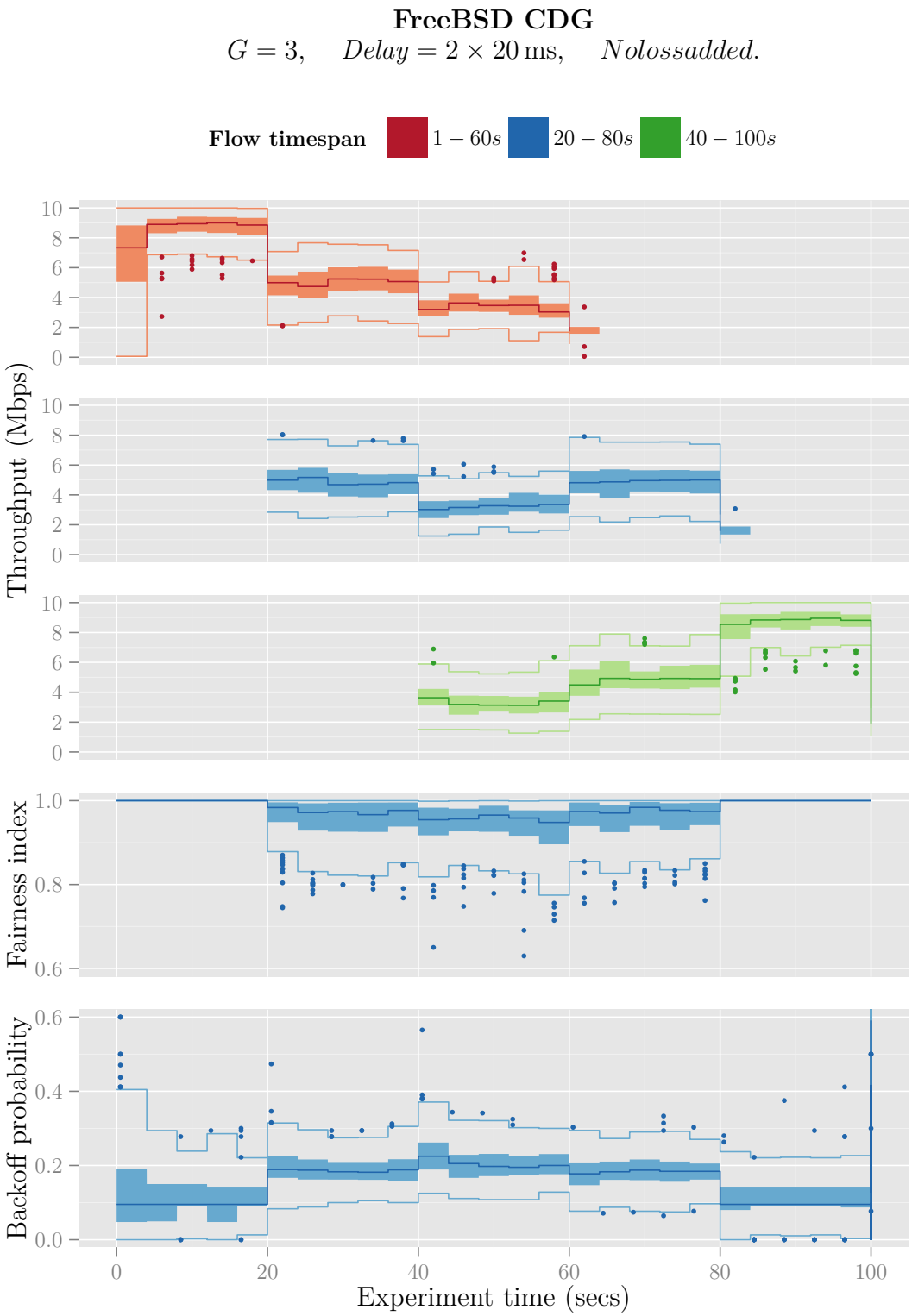


Figure A.18: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

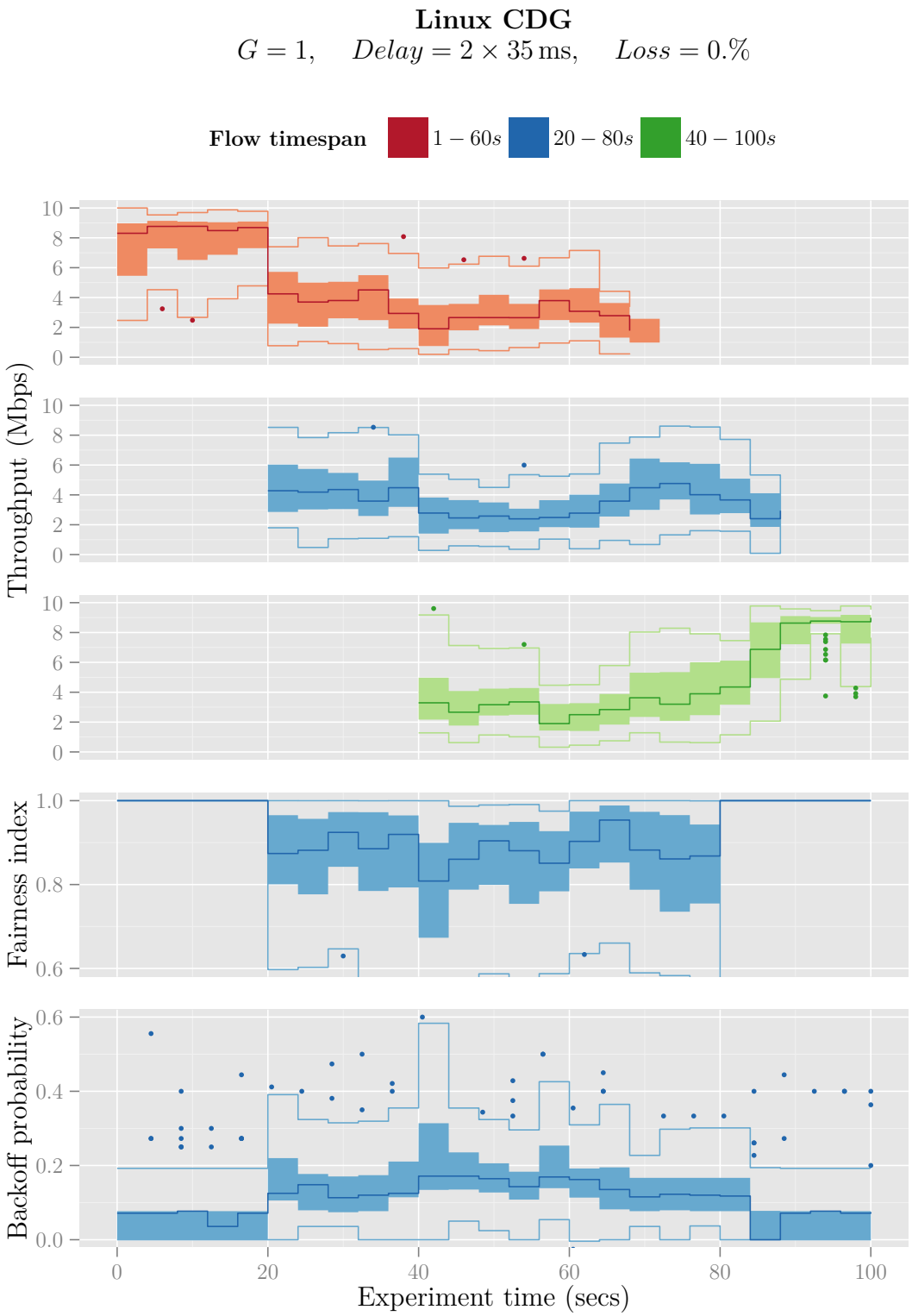


Figure A.19: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

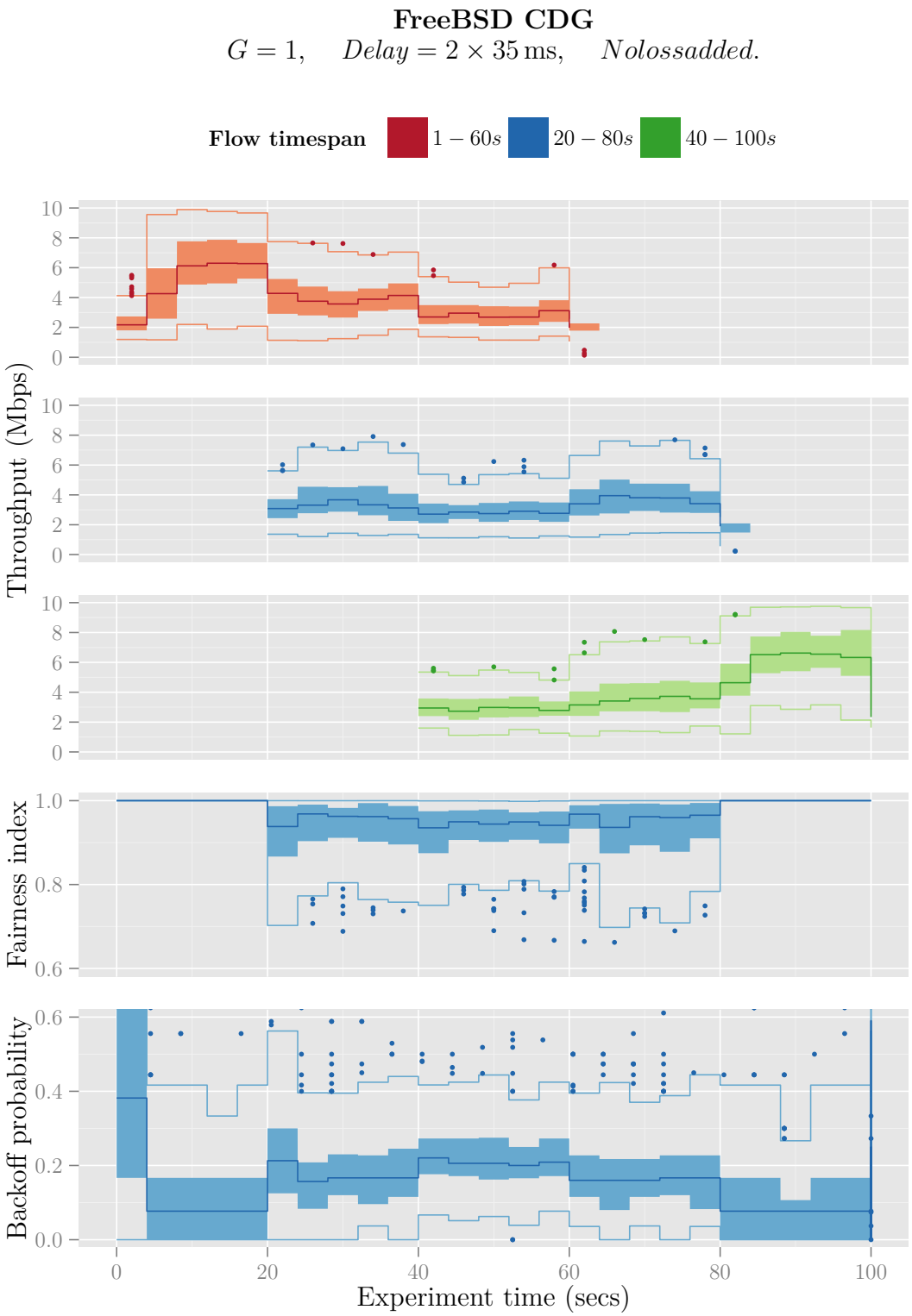


Figure A.20: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

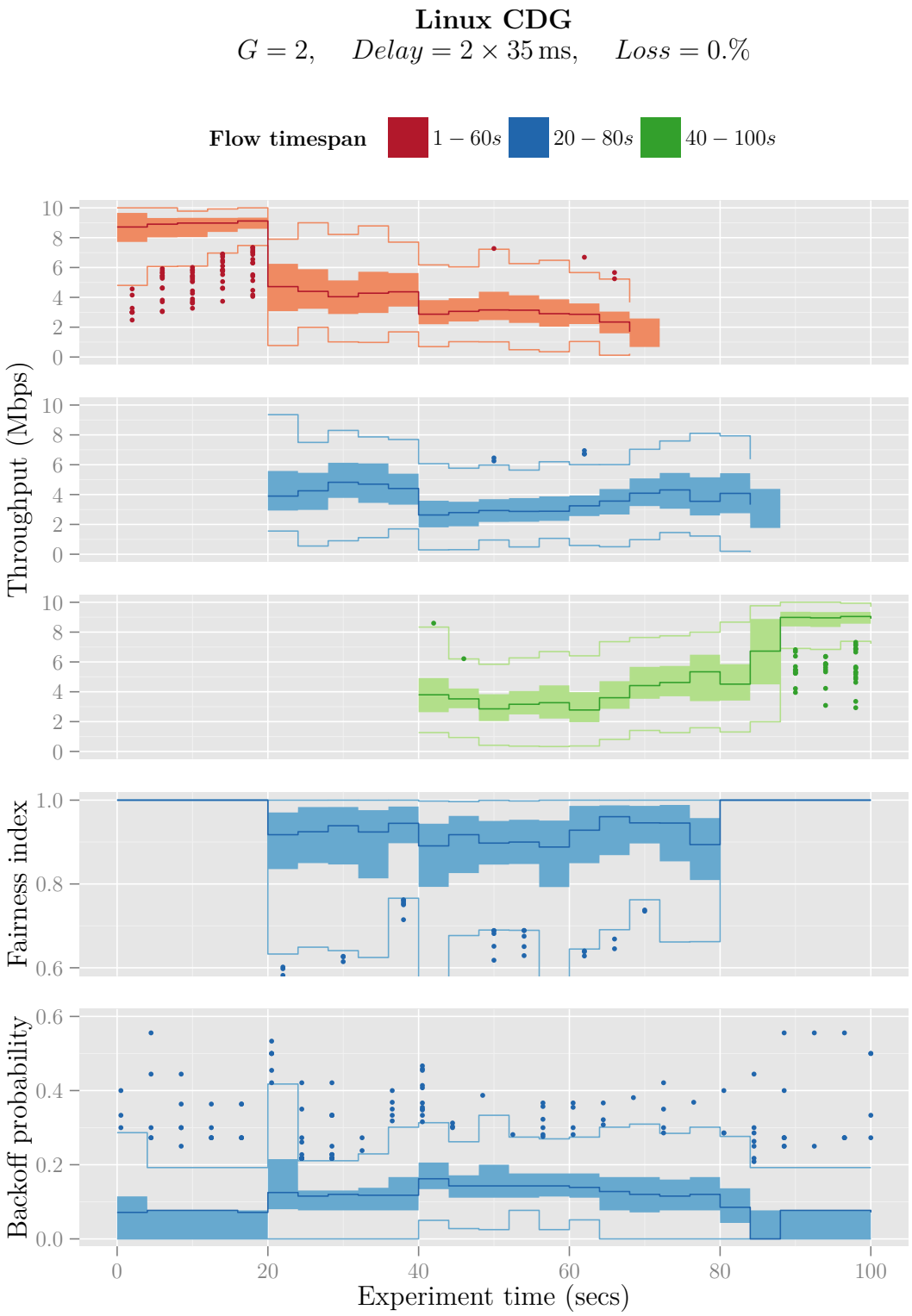


Figure A.21: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

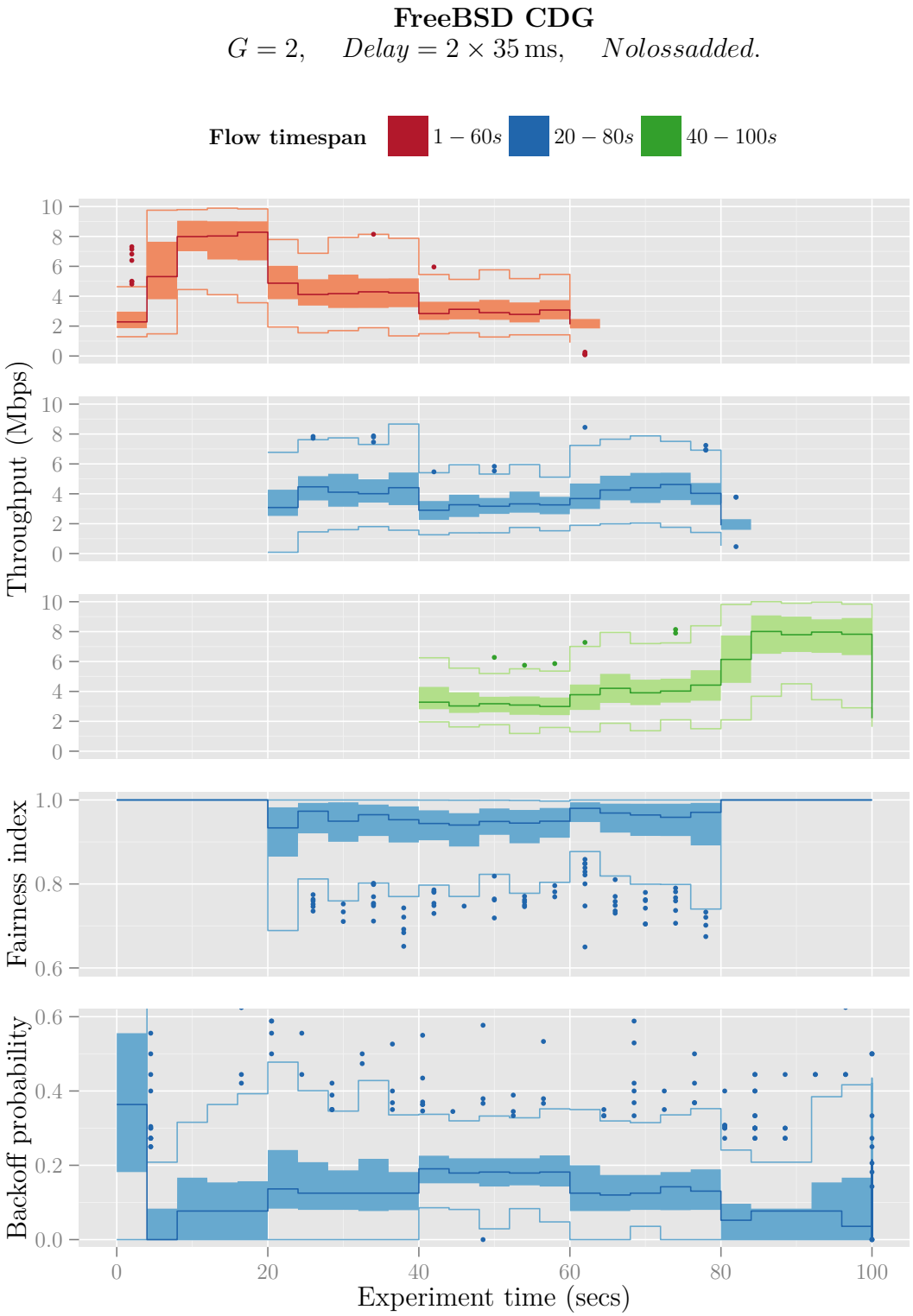


Figure A.22: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

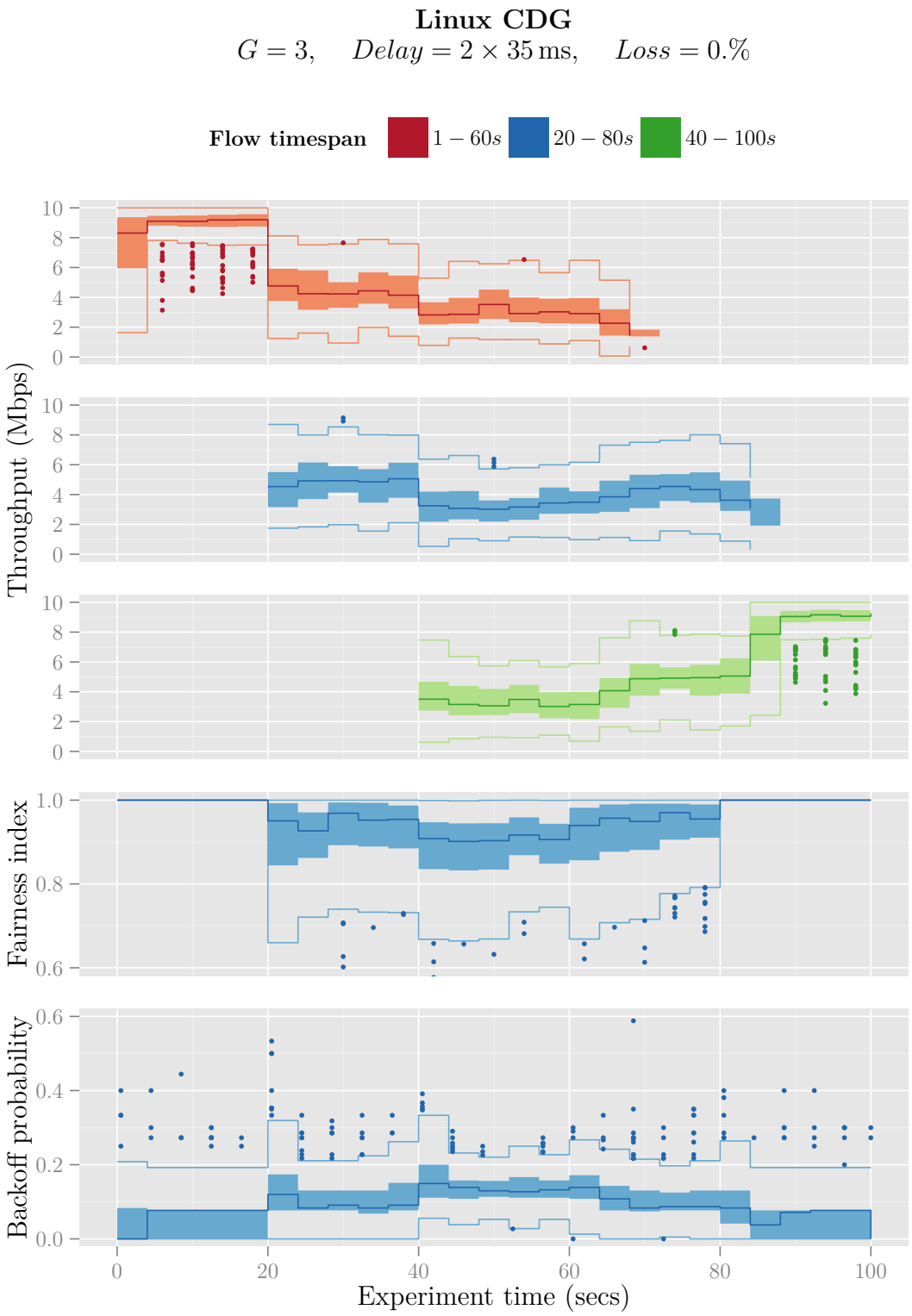


Figure A.23: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

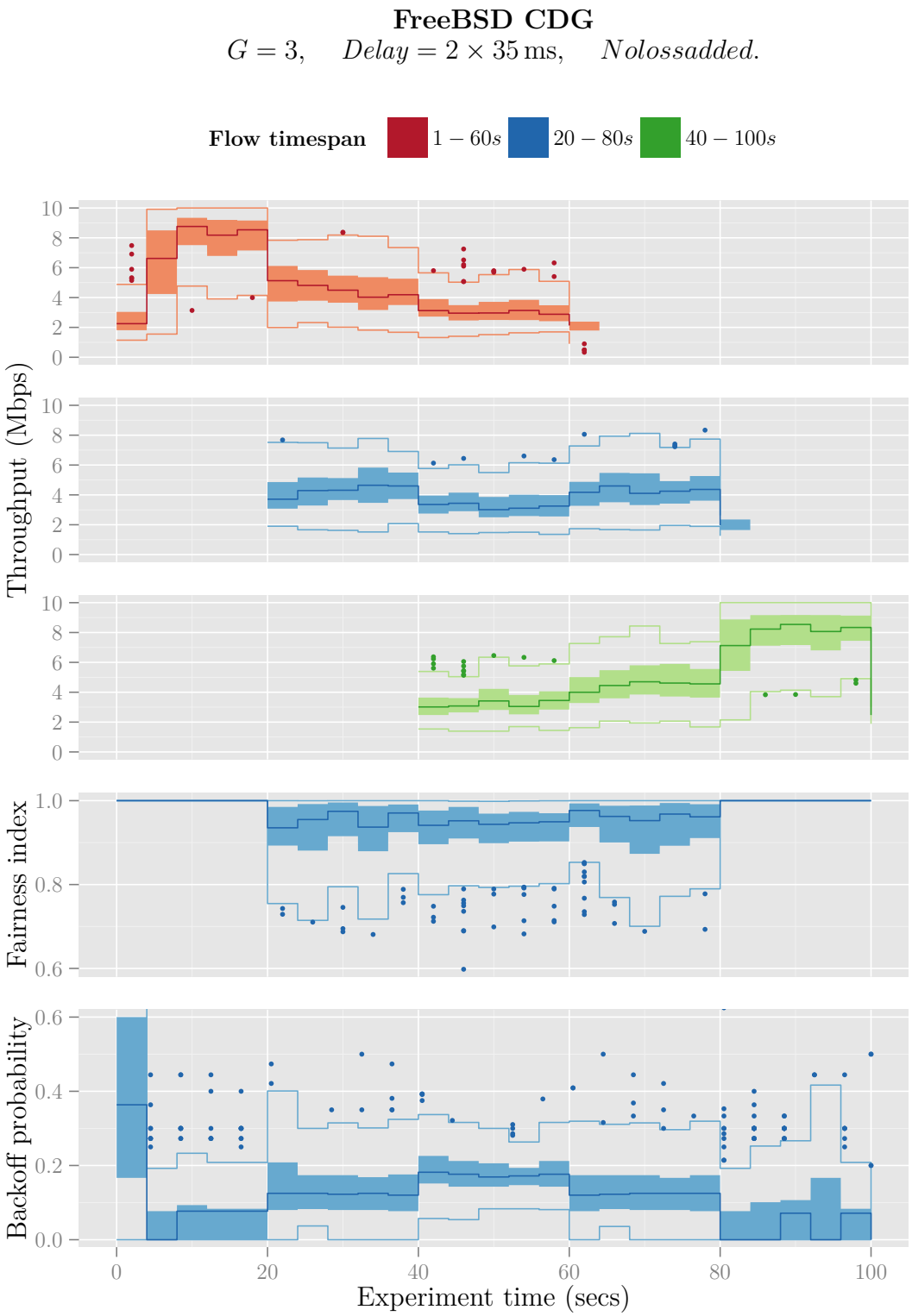


Figure A.24: Data calculated at discrete 1 s intervals. Summarised in 4 s intervals with median, interquartile range, whiskers and outliers. Throughput as transferred bytes per second. Backoff probability as number of backoffs divided by number of possible backoffs. No averaging or smoothing applied.

Appendix B

Linux implementation of CDG

```
/*
 * CAIA Delay-Gradient (CDG) congestion control
 *
 * This implementation is based on the paper:
 *   D.A. Hayes and G. Armitage. "Revisiting TCP congestion control using
 *   delay gradients." In Networking 2011, pages 328-341. Springer, 2011.
 *
 * For background traffic, disable coexistence heuristics using parameters
 * use_shadow=0 ineffective_thresh=0.
 *
 * Notable differences from paper and FreeBSD patch:
 *   o Add toggle for shadow window mechanism. Suggested by David Hayes.
 *   o Add toggle for non-congestion loss tolerance.
 *   o Scaling parameter G is changed to a backoff factor;
 *     conversion is given by: backoff_factor = 1000/G.
 *   o Clear shadow window on ambiguity between full and empty queue.
 *   o Limit shadow window when application limited.
 *   o Re-initialize shadow window on cwnd restart.
 *   o Infer full queue on ECN signal.
 *   o More accurate e^-x.
 */
#include <linux/kernel.h>
#include <linux/random.h>
#include <linux/module.h>
#include <net/tcp.h>

static  int window __read_mostly = 8;
static bool use_shadow __read_mostly = true;
static bool use_tolerance __read_mostly;
static uint backoff_beta __read_mostly = 0.70 * 1024;
static uint backoff_factor __read_mostly = 333;
static uint ineffective_thresh __read_mostly = 5;
static uint ineffective_hold __read_mostly = 5;

module_param(window, int, 0444);
MODULE_PARM_DESC(window, "moving average window width (power of two)");
module_param(use_shadow, bool, 0644);
MODULE_PARM_DESC(use_shadow, "use shadow window heuristic");
module_param(use_tolerance, bool, 0644);
MODULE_PARM_DESC(use_tolerance, "use loss tolerance heuristic");
module_param(backoff_beta, uint, 0444);
MODULE_PARM_DESC(backoff_beta, "backoff beta (1-1024)");
```

```
module_param(backoff_factor, uint, 0444);
MODULE_PARM_DESC(backoff_factor, "backoff probability scale factor (1-65535)");
module_param(ineffective_thresh, uint, 0644);
MODULE_PARM_DESC(ineffective_thresh, "ineffective backoff threshold");
module_param(ineffective_hold, uint, 0644);
MODULE_PARM_DESC(ineffective_hold, "ineffective backoff hold time");

struct minmax {
        union {
                struct {
                        s32 min;
                        s32 max;
                };
                u64 v64;
        };
};

enum cdg_state {
        CDG_UNKNOWN = 0,
        CDG_FULL    = 0,
        CDG_NONFULL = 1,
};

struct cdg {
        struct minmax rtt;
        struct minmax rtt_prev;
        struct minmax gsum;
        struct minmax *gradients;
        enum cdg_state state;
        u32 rtt_seq;
        u32 shadow_wnd;
        u32 loss_cwnd;
        uint tail;
        uint backoff_cnt;
        uint delack;
        bool ecn_ce;
};

/**
 * nexp_u32 - negative base-e exponential
 * @ux: x in units of micro
 *
 * Returns exp(ux * -1e-6) * U32_MAX.
 */
static u32 __pure nexp_u32(u32 ux)
{
        static const u16 v[] = {
                /* exp(-x)*65536-1 for x = 0, 0.000256, 0.000512, ... */
                65535,
                65518, 65501, 65468, 65401, 65267, 65001, 64470, 63422,
                61378, 57484, 50423, 38795, 22965, 8047,  987,   14,
        };
        u64 res;
        u32 msb = ux >> 8;
        int i;

        /* Cut off when ux >= 2^24 (actual result is <= 222/U32_MAX). */
        if (msb > U16_MAX)
                return 0;

        /* Scale first eight bits linearly: */
        res = U32_MAX - (ux & 0xff) * (U32_MAX / 1000000);
```

```
        /* Obtain e^(x + y + ...) by computing e^x * e^y * ...: */
        for (i = 1; msb; i++, msb >>= 1) {
                u64 y = v[i & -(msb & 1)] + 1ULL;

                res = (res * y) >> 16;
        }

        return (u32)res;
}

static s32 tcp_cdg_grad(struct sock *sk)
{
        struct cdg *ca = inet_csk_ca(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        s32 grad = 0;

        if (ca->rtt_prev.v64) {
                s32 gmin = ca->rtt.min - ca->rtt_prev.min;
                s32 gmax = ca->rtt.max - ca->rtt_prev.max;
                s32 gmin_s;
                s32 gmax_s;

                ca->gsum.min += gmin - ca->gradients[ca->tail].min;
                ca->gsum.max += gmax - ca->gradients[ca->tail].max;
                ca->gradients[ca->tail].min = gmin;
                ca->gradients[ca->tail].max = gmax;
                ca->tail = (ca->tail + 1) & (window - 1);

                /* We keep sums to ignore gradients during CWR;
                 * smoothed gradients otherwise simplify to:
                 * (rtt_latest - rtt_oldest) / window.
                 */
                gmin_s = DIV_ROUND_CLOSEST(ca->gsum.min, window);
                gmax_s = DIV_ROUND_CLOSEST(ca->gsum.max, window);

                /* Only use smoothed gradients in CA: */
                if (tp->snd_cwnd > tp->snd_ssthresh) {
                        grad = gmin_s > 0 ? gmin_s : gmax_s;
                } else {
                        /* Prefer unsmoothed gradients in slow start. */
                        grad = gmin > 0 ? gmin : gmin_s;
                        if (grad < 0)
                                grad = gmax > 0 ? gmax : gmax_s;
                }

                if (gmin_s > 0 && gmax_s <= 0)
                        ca->state = CDG_FULL;
                else if ((gmin_s > 0 && gmax_s > 0) || gmax_s < 0)
                        ca->state = CDG_NONFULL;

                /* Empty queue: */
                if (gmin_s >= 0 && gmax_s < 0)
                        ca->shadow_wnd = 0;
                /* Backoff was effectual: */
                if (gmin_s < 0 || gmax_s < 0)
                        ca->backoff_cnt = 0;
        }

        ca->rtt_prev = ca->rtt;
        ca->rtt.v64 = 0;
        return grad;
```

```
}

static int tcp_cdg_backoff(struct sock *sk, s32 grad)
{
        struct cdg *ca = inet_csk_ca(sk);
        struct tcp_sock *tp = tcp_sk(sk);

        if (grad <= 0 || prandom_u32() <= nexp_u32(grad * backoff_factor))
                return 0;

        ca->shadow_wnd = max(ca->shadow_wnd, tp->snd_cwnd);
        ca->backoff_cnt++;
        if (ca->backoff_cnt > ineffective_thresh && ineffective_thresh) {
                if (ca->backoff_cnt >= (ineffective_thresh + ineffective_hold))
                        ca->backoff_cnt = 0;
                return 0;
        }

        /* reset TLP and prohibit cwnd undo: */
        tp->tlp_high_seq = 0;
        tp->prior_ssthresh = 0;

        /* set PRR target and enter CWR: */
        tp->snd_ssthresh = max(2U, (tp->snd_cwnd * backoff_beta) >> 10U);
        tp->prr_delivered = 0;
        tp->prr_out = 0;
        tp->prior_cwnd = tp->snd_cwnd;
        tp->high_seq = tp->snd_nxt;

        tcp_set_ca_state(sk, TCP_CA_CWR);
        return 1;
}

/* Not called in CWR or Recovery state. */
static void tcp_cdg_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
        struct cdg *ca = inet_csk_ca(sk);
        struct tcp_sock *tp = tcp_sk(sk);

        if (unlikely(!ca->gradients))
                return tcp_reno_cong_avoid(sk, ack, acked);

        /* Measure filtered gradients once per RTT: */
        if (after(ack, ca->rtt_seq) && ca->rtt.v64) {
                s32 grad = tcp_cdg_grad(sk);

                ca->rtt_seq = tp->snd_nxt;

                if (tcp_cdg_backoff(sk, grad))
                        return;
        } else if (tp->snd_cwnd > tp->snd_ssthresh) {
                /* In CA, synchronize cwnd growth with delay gradients. */
                return;
        }

        if (!tcp_is_cwnd_limited(sk)) {
                ca->shadow_wnd = min(ca->shadow_wnd, tp->snd_cwnd);
                return;
        }

        if (tp->snd_cwnd <= tp->snd_ssthresh)
                tcp_slow_start(tp, acked);
```

```
        else if (tp->snd_cwnd < tp->snd_cwnd_clamp)
                tp->snd_cwnd++;

        if (ca->shadow_wnd && ca->shadow_wnd < tp->snd_cwnd_clamp)
                ca->shadow_wnd = max(tp->snd_cwnd, ca->shadow_wnd + 1);
}

static void tcp_cdg_acked(struct sock *sk, u32 num_acked, s32 rtt_us)
{
        struct cdg *ca = inet_csk_ca(sk);

        if (rtt_us <= 0)
                return;

        /* A heuristic for filtering delayed ACKs, adapted from:
         * D.A. Hayes. "Timing enhancements to the FreeBSD kernel to support
         * delay and rate based TCP mechanisms." TR 100219A. CAIA, 2010.
         *
         * Assume num_acked == 0 indicates RTT measurement from SACK.
         */
        if (num_acked == 1 && ca->delack) {
                /* A delayed ACK is only used for the minimum if it is
                 * provenly lower than an existing non-zero minimum.
                 */
                ca->rtt.min = min(ca->rtt.min, rtt_us);
                ca->delack--;
                return;
        } else if (num_acked > 1 && ca->delack < 5) {
                ca->delack++;
        }

        ca->rtt.min = min_not_zero(ca->rtt.min, rtt_us);
        ca->rtt.max = max(ca->rtt.max, rtt_us);
}

static u32 tcp_cdg_ssthresh(struct sock *sk)
{
        struct cdg *ca = inet_csk_ca(sk);
        struct tcp_sock *tp = tcp_sk(sk);

        if (unlikely(!ca->gradients))
                return tcp_reno_ssthresh(sk);

        ca->loss_cwnd = tp->snd_cwnd;

        if (use_tolerance && ca->state == CDG_NONFULL && !ca->ecn_ce)
                return tp->snd_cwnd;

        ca->shadow_wnd >>= 1;
        if (!use_shadow)
                return max(2U, tp->snd_cwnd >> 1);
        return max3(2U, ca->shadow_wnd, tp->snd_cwnd >> 1);
}

static u32 tcp_cdg_undo_cwnd(struct sock *sk)
{
        struct cdg *ca = inet_csk_ca(sk);

        return max(tcp_sk(sk)->snd_cwnd, ca->loss_cwnd);
}

static void tcp_cdg_cwnd_event(struct sock *sk, const enum tcp_ca_event ev)
```

```
{
        struct cdg *ca = inet_csk_ca(sk);
        struct tcp_sock *tp = tcp_sk(sk);
        struct minmax *gradients;

        switch (ev) {
        case CA_EVENT_ECN_NO_CE:
                ca->ecn_ce = false;
                break;
        case CA_EVENT_ECN_IS_CE:
                ca->ecn_ce = true;
                ca->state = CDG_UNKNOWN;
                break;
        case CA_EVENT_CWND_RESTART:
                gradients = ca->gradients;
                if (gradients)
                        memset(gradients, 0, window * sizeof(gradients[0]));
                memset(ca, 0, sizeof(*ca));

                ca->gradients = gradients;
                ca->rtt_seq = tp->snd_nxt;
                ca->shadow_wnd = tp->snd_cwnd;
                break;
        case CA_EVENT_COMPLETE_CWR:
                ca->state = CDG_UNKNOWN;
                ca->rtt_seq = tp->snd_nxt;
                ca->rtt_prev = ca->rtt;
                ca->rtt.v64 = 0;
                break;
        default:
                break;
        }
}

static void tcp_cdg_init(struct sock *sk)
{
        struct cdg *ca = inet_csk_ca(sk);
        struct tcp_sock *tp = tcp_sk(sk);

        /* May fail. Not allocating from emergency pools. */
        ca->gradients = kcalloc(window, sizeof(ca->gradients[0]), GFP_NOWAIT);
        ca->shadow_wnd = tp->snd_cwnd;
        ca->rtt_seq = tp->snd_nxt;
}

static void tcp_cdg_release(struct sock *sk)
{
        struct cdg *ca = inet_csk_ca(sk);

        kfree(ca->gradients);
}

struct tcp_congestion_ops tcp_cdg __read_mostly = {
        .cong_avoid = tcp_cdg_cong_avoid,
        .cwnd_event = tcp_cdg_cwnd_event,
        .pkts_acked = tcp_cdg_acked,
        .undo_cwnd = tcp_cdg_undo_cwnd,
        .ssthresh = tcp_cdg_ssthresh,
        .release = tcp_cdg_release,
        .init = tcp_cdg_init,
        .owner = THIS_MODULE,
        .name = "cdg",
```

```
};

static int __init tcp_cdg_register(void)
{
        window = clamp(window, 1, 256);
        backoff_beta = clamp(backoff_beta, 1U, 1024U);
        backoff_factor = clamp(backoff_factor, 1U, 65535U);

        if (!is_power_of_2(window))
                return -EINVAL;

        BUILD_BUG_ON(sizeof(struct cdg) > ICSK_CA_PRIV_SIZE);
        tcp_register_congestion_control(&tcp_cdg);
        return 0;
}

static void __exit tcp_cdg_unregister(void)
{
        tcp_unregister_congestion_control(&tcp_cdg);
}

module_init(tcp_cdg_register);
module_exit(tcp_cdg_unregister);
MODULE_AUTHOR("Kenneth Klette Jonassen");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP CDG");
```

Appendix C

Kernel patches

```
commit 932eb7638ad7d9145620178992044b5e87356969
Author: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
Date:   Thu Jan 29 20:08:03 2015 +0100

    tcp: use SACK RTTs for CC

    Current behavior only passes RTTs from sequentially acked data to CC.

    If sender gets a combined ACK for segment 1 and SACK for segment 3, then the
    computed RTT for CC is the time between sending segment 1 and receiving SACK
    for segment 3.

    Pass the minimum computed RTT from any acked data to CC, i.e. time between
    sending segment 3 and receiving SACK for segment 3.

    Signed-off-by: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
    Signed-off-by: David S. Miller <davem@davemloft.net>


commit 3d0d26c7976bf190c3f1d2efbc31462db8246bc0
Author: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
Date:   Sat Apr 11 02:17:49 2015 +0200

    tcp: fix bogus RTT for CC when retransmissions are acked

    Since retransmitted segments are not used for RTT estimation, previously
    SACKed segments present in the rtx queue are used. This estimation can be
    several times larger than the actual RTT. When a cumulative ack covers both
    previously SACKed and retransmitted segments, CC may thus get a bogus RTT.

    Such segments previously had an RTT estimation in tcp_sacktag_one(), so it
    seems reasonable to not reuse them in tcp_clean_rtx_queue() at all.

    Afaik, this has had no effect on SRTT/RTO because of Karn's check.

    Signed-off-by: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
    Acked-by: Neal Cardwell <ncardwell@google.com>
    Tested-by: Neal Cardwell <ncardwell@google.com>
    Acked-by: Yuchung Cheng <ycheng@google.com>
    Signed-off-by: David S. Miller <davem@davemloft.net>
```

```
commit 196da974758550a3933c8b0244ef98148df10552
Author: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
Date:   Fri May 1 01:10:57 2015 +0200

    tcp: move struct tcp_sacktag_state to tcp_ack()

    Later patch passes two values set in tcp_sacktag_one() to
    tcp_clean_rtx_queue(). Prepare passing them via struct tcp_sacktag_state.

    Acked-by: Yuchung Cheng <ycheng@google.com>
    Cc: Eric Dumazet <edumazet@google.com>
    Signed-off-by: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
    Signed-off-by: David S. Miller <davem@davemloft.net>



commit 31231a8a873026410eab438c5757430546a517d1
Author: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
Date:   Fri May 1 01:10:58 2015 +0200

    tcp: improve RTT from SACK for CC

    tcp_sacktag_one() always picks the earliest sequence SACKed for RTT.
    This might not make sense for congestion control in cases where:

      1. ACKs are lost, i.e. a SACK following a lost SACK covers both
         new and old segments at the receiver.
      2. The receiver disregards the RFC 5681 recommendation to immediately
         ACK out-of-order segments.

    Give congestion control a RTT for the latest segment SACKed, which is the
    most accurate RTT estimate, but preserve the conservative RTT for RTO.

    Removes the call to skb_mstamp_get() in tcp_sacktag_one().

    Cc: Yuchung Cheng <ycheng@google.com>
    Cc: Eric Dumazet <edumazet@google.com>
    Signed-off-by: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
    Acked-by: Yuchung Cheng <ycheng@google.com>
    Signed-off-by: David S. Miller <davem@davemloft.net>



commit 138998fdd12e7362756e158d00856a2aabd5f0c1
Author: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
Date:   Fri May 1 01:10:59 2015 +0200

    tcp: invoke pkts_acked hook on every ACK

    Invoking pkts_acked is currently conditioned on FLAG_ACKED:
    receiving a cumulative ACK of new data, or ACK with SYN flag set.

    Remove this condition so that CC may get RTT measurements from all SACKs.

    Cc: Yuchung Cheng <ycheng@google.com>
    Cc: Eric Dumazet <edumazet@google.com>
    Cc: Neal Cardwell <ncardwell@google.com>
    Signed-off-by: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
    Signed-off-by: David S. Miller <davem@davemloft.net>
```

Other

```
commit 3725a269815ba6dbb415feddc47da5af7d1fac58
Author: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
Date:   Tue Feb 3 17:49:18 2015 +0100

    pkt_sched: fq: avoid hang when quantum 0

    Configuring fq with quantum 0 hangs the system, presumably because of a
    non-interruptible infinite loop. Either way quantum 0 does not make sense.

    Reproduce with:
    sudo tc qdisc add dev lo root fq quantum 0 initial_quantum 0
    ping 127.0.0.1

    Signed-off-by: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
    Acked-by: Eric Dumazet <edumazet@google.com>
    Signed-off-by: David S. Miller <davem@davemloft.net>
```

Unmerged

```
Author: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
Date:   Tue Feb 3 19:48:00 2015 +0100

    pkt_sched: fq: avoid artificial bursts for clocked flows

    Current pacing behavior always throttle flows for a time equal to one full
    quantum, starting when a flow exhausts its credit. This is only optimal for
    bursty traffic that consumes the entire credit in one sitting.

    For flows with small packets, this throttling behavior can cause packets
    to artificially queue and transmit in bursts, even when their sending rate
    is well below their pacing rate. There is a refill mechanism in fq_enqueue()
    that counteracts this in some cases, but it only works when flows space
    their packets further apart than the flow refill delay.

    Keep track of the time a flows credit was last filled, and use this to
    approximate a full credit refill when one quantum of time passes. This is
    a more fine-grained approach than the refill heuristic in fq_enqueue(),
    which is removed by the next patch in this series.

    Since calls to ktime are expensive, time_credit_filled is only set correctly
    on dequeue. For new flows, set time_credit_filled to zero and anticipate
    dequeue to refill one quantum without throttling. This approach requires
    that initial_quantum >= quantum.

    Increases memory footprint from 104 to 112 bytes per flow.

    Signed-off-by: Kenneth Klette Jonassen <kennetkl@ifi.uio.no>
    ---
    V2: avoids ktime_get_ns() on enqueue, as suggested by Eric Dumazet.
```

Appendix D

Experiment details

Linux sender Quad-core Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
Kernel: Linux bestemor 4.1.0-rc3+ #68 SMP Sun May 17 17:12:57 CEST 2015 x86_64 GNU/Linux

Linux router Dual-core Intel(R) Core(TM)2 Duo CPU E8200 @ 2.66GHz
Kernel: Linux router 4.1.0-rc1+ #67 SMP Fri May 1 23:52:38 CEST 2015 x86_64 GNU/Linux

Linux receiver Dual-core Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz
Kernel: 3.19.0+web10g #6 SMP Sun Mar 1 18:07:08 CET 2015 x86_64 GNU/Linux

FreeBSD sender Dual-core Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz
Kernel: 10.0-RELEASE FreeBSD 10.0-RELEASE #0 r260789: Thu Jan 16 22:34:59 UTC 2014 amd64

References

[1] M. Alizadeh, A. Greenberg, D. Maltz, et al. Data center TCP (DCTCP). *ACM SIGCOMM*, 41(4):63–74, 2011. 15, 23, 74

[2] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009. 2, 7, 8, 10, 20

[3] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. *SIGCOMM Comput. Commun. Rev.*, 34(4):281–292, August 2004. 12, 13, 14

[4] G. Armitage et al. Delay-gradient TCP. 84, Vancouver, July 2012. IETF. A talk presented at the ICCRG (audio: ietf84-georgiaa-20120730-1300-pm1.mp3). 15, 23, 24, 25, 27, 28, 73

[5] G. Armitage and N. Khademi. Using delay-gradient TCP for multimedia-friendly "background" transport in home networks. In *Local Computer Networks*, pages 509–515. IEEE, Oct 2013. 42, 71

[6] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao. Overview and Principles of Internet Traffic Engineering. RFC 3272 (Informational), May 2002. Updated by RFC 5462. 13, 16

[7] F. Baker. Requirements for IP Version 4 Routers. RFC 1812 (Proposed Standard), June 1995. Updated by RFCs 2644, 6633. 6, 22

[8] F. Baker. [end2end-interest] deflating excessive buffers. Private communication: http://goo.gl/g4ndh2 (postel.org), October 2014. 11

[9] F. Baker and G. Fairhurst. IETF recommendations regarding active queue management, January 2015. 15

[10] S. Bjørnstad. Introduction to optical networking. In *INF5050 - Routing and protocols in the Internet*. University of Oslo, Norway, January 2015. Lecture. 12

[11] B. Braden, D. Clark, J. Crowcroft, et al. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309 (Informational), April 1998. Updated by RFC 7141. 13, 14, 15

[12] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Internet standard), October 1989. Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864. 5, 6, 20

[13] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), March 1997. 10

[14] L. Brakmo and L. Peterson. TCP vegas: End to end congestion avoidance on a global internet. *Selected Areas in Communications, IEEE Journal on*, 13(8):1465–1480, 1995. 20, 21

[15] B. Briscoe. Flow rate fairness: Dismantling a religion. *ACM SIGCOMM Computer Communication Review*, 37(2):63–74, April 2007. 10

[16] B. Briscoe, G. Corlianó, and B. Strulo. How to Build a Virtual Queue from Two Leaky Buckets (and why one is not enough). Technical Report TR-DES8-2011-001, BT, April 2012. 17

[17] B. Briscoe, R. Scheffenegger, and M. Kuehlewind. More accurate ECN feedback in TCP. draft-kuehlewind-tcpm-accurate-ecn-03, July 2014. Expires January 3rd, 2015. 23, 74

[18] B. Briscoe, R. Scheffenegger, and M. Kuehlewind. Problem statement and requirements for a more accurate ECN feedback. draft-ietf-tcpm-accecn-reqs-08, March 2015. Expires September 10th, 2015. 23, 74

[19] R. Bush and D. Meyer. Some Internet Architectural Guidelines and Philosophy. RFC 3439 (Informational), December 2002. 13

[20] H. Chao. Next generation routers. In *Proceedings of the IEEE*, pages 1518–1558, 2002. 12

[21] Y. Cheng et al. Recent advancements in linux tcp congestion control. 88, Vancouver, November 2013. IETF. A talk presented at the ICCRG. 45, 71

[22] U. Drepper. What every programmer should know about memory. 2007. 40

[23] S. Floyd. Congestion Control Principles. RFC 2914 (Best Current Practice), September 2000. Updated by RFC 7141. 7

[24] S. Floyd. Metrics for the Evaluation of Congestion Control Mechanisms. RFC 5166 (Informational), March 2008. 9, 10, 47, 69

[25] S. Floyd and M. Allman. Specifying New Congestion Control Algorithms. RFC 5033 (Best Current Practice), August 2007. 24, 47, 69

[26] Y. Ganjali and N. McKeown. Update on buffer sizing in internet routers. *ACM SIGCOMM Computer Communication Review*, 36(5):67–70, 2006. 12, 13, 14

[27] J. Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the internet. *Commun. ACM*, 55(1):57–65, January 2012. 11, 12, 13, 14

[28] F. Gont. Deprecation of ICMP Source Quench Messages. RFC 6633 (Proposed Standard), May 2012. 22

[29] S. Ha and I. Rhee. Hybrid slow start for high-bandwidth and long-distance networks. In *Proceedings of PFLDnet*. PFLDnet, 2008. 45, 71

[30] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008. 2, 60

[31] M. Handley, J. Padhye, and S. Floyd. TCP Congestion Window Validation. RFC 2861 (Experimental), June 2000. 34, 42

[32] D. Hayes. Timing enhancements to the FreeBSD kernel to support delay and rate based TCP mechanisms. Technical Report TR 100219A, Centre for Advanced Internet Architectures, 2010. 25, 62, 70

[33] D. Hayes and G. Armitage. Improved coexistence and loss tolerance for delay based TCP congestion control. In *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, pages 24–31, Oct 2010. 3, 28

[34] D. Hayes and G. Armitage. Revisiting TCP congestion control using delay gradients. In *NETWORKING 2011*, pages 328–341. Springer, 2011. 2, 3, 19, 20, 23, 24, 25, 26, 28, 39, 45, 52

[35] D. Hayes, D. Ros, L. Andrew, and S. Floyd. Common TCP evaluation suite, July 2014. Expired January 5th, 2015. 47, 69

[36] D. Hayes, L. Stewart, and G. Armitage. FreeBSD implementation of TCP Vegas. FreeBSD official git tree fec0e548bc: http://goo.gl/iSTP11 (github.com), 2011. 21

[37] D. Hayes, L. Stewart, and G. Armitage. FreeBSD implementation of TCP CDG. FreeBSD official git tree 1920a78bd9: http://goo.gl/pPNSBB (github.com), 2013. 24, 25, 28, 43

[38] S. Hemminger. TCP infrastructure split out. Mailing list: http://goo.gl/xYYWml (gmane.org), 2005. 34

[39] S. Hemminger et al. Linux implementation of TCP Vegas. Linux official git tree 688d1945bc: http://goo.gl/7hGK53 (github.com), 2014. 21

[40] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582 (Proposed Standard), April 2012. 10

[41] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988. 8, 18

[42] V. Jacobson. Dynamic congestion avoidance / control (long message). Private communication: http://ee.lbl.gov/tcp.html, February 1988. 2, 18, 22

[43] V. Jacobson. A rant on queues. A talk presented at MIT Lincoln Labs, July 2006. 15

[44] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992. Obsoleted by RFC 7323. 18, 20

[45] L. Kleinrock. *Queueing systems: volume 2: computer applications*, volume 82. John Wiley & Sons New York, 1976. 11

[46] M. Mathis, N. Dukkipati, and Y. Cheng. Proportional Rate Reduction for TCP. RFC 6937 (Experimental), May 2013. 43

[47] G. McCullagh and D.J. Leith. Delay-based congestion control: Sampling and correlation issues revisited. *Hamilton Institute—National University of Ireland, Maynooth, Tech. Rep*, 2008. 19

[48] J. Nagle. On Packet Switches With Infinite Storage. RFC 970, December 1985. 12

[49] K. Nichols and V. Jacobson. Controlling queue delay. *Commun. ACM*, 55(7):42–50, July 2012. 13, 14

[50] D. Papadimitriou, M. Welzl, M. Scharf, and B. Briscoe. Open Research Issues in Internet Congestion Control. RFC 6077 (Informational), February 2011. 2, 18, 23

[51] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP Implementation Problems. RFC 2525 (Informational), March 1999. 20

[52] A. Petlund. *Improving latency for interactive, thin-stream applications over reliable transport*. PhD thesis, University of Oslo, 2009. 12

[53] J. Postel. Internet Control Message Protocol. RFC 792 (Internet standard), September 1981. Updated by RFCs 950, 4884, 6633, 6918. 22

[54] J. Postel. Internet Protocol. RFC 791 (Internet standard), September 1981. Updated by RFCs 1349, 2474, 6864. 6, 7

[55] R. Prasad, M. Jain, and C. Dovrolis. On the effectiveness of delay-based congestion avoidance. In *Proceedings of PFLDnet*. PFLDnet, 2004. 19

[56] G. Raina, D. Towsley, and D. Wischik. Part II: Control theory for buffer sizing. *SIGCOMM Comput. Commun. Rev.*, 35(3):79–82, July 2005. 12

[57] K. Ramakrishnan and S. Floyd. A Proposal to add Explicit Congestion Notification (ECN) to IP. RFC 2481 (Experimental), January 1999. Obsoleted by RFC 3168. 22

[58] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001. Updated by RFCs 4301, 6040. 12, 22, 23

[59] K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer. *SIGCOMM Comput. Commun. Rev.*, 18(4):303–313, August 1988. 22

[60] D. Ros and Michael Welzl. Assessing ledbat's delay impact. *Communications Letters, IEEE*, 17(5):1044–1047, 2013. 21

[61] R. Scheffenegger, M. Kuehlewind, and B. Trammell. Additional negotiation in the TCP timestamp option field during the TCP handshake, October 2012. Expired April 25th, 2013. 72

[62] R. Scheffenegger, M. Kuehlewind, and B. Trammell. Encoding of time intervals for the TCP timestamp option, July 2013. Expired January 16th, 2014. 72

[63] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). RFC 6817 (Experimental), December 2012. 2, 21, 71

[64] S. Smith. *Digital Signal Processing: A Practical Guide for Engineers and Scientists*. Demystifying technology series. Elsevier Science, 2013. 18, 25, 67

[65] N. Spring, D. Wetherall, and D. Ely. Robust Explicit Congestion Notification (ECN) Signaling with Nonces. RFC 3540 (Experimental), June 2003. 23

[66] T. Szigeti, C. Hattingh, R. Barton, et al. *End-To-End QoS Network Design: Quality of Service for Rich-Media and Cloud Networks*. Cisco Press networking technology series. Cisco Press, 2013. 12, 13, 15, 16, 17

[67] A. Tanenbaum and D. Wetherall. *Computer Networks*. Pearson, 2011. 5, 7, 17, 18

[68] L. Torvalds et al. net/ipv4/tcp_input.c. Linux official git tree: https://goo.gl/S87eU0 (github.com), 2015. 20

[69] G. Vu-Brugier, R.S. Stanojevic, Douglas J. Leith, et al. A critique of recently proposed buffer-sizing strategies. *ACM SIGCOMM Computer Communication Review*, 37(1):43–48, 2007. 14

[70] M. Welzl. *Network Congestion Control: Managing Internet Traffic*. John Wiley & Sons, 2005. 2, 7, 10, 14, 17, 18, 19

[71] D. Wischik and N. McKeown. Part I: Buffer sizes for core routers. *SIGCOMM Comput. Commun. Rev.*, 35(3):75–78, July 2005. 12, 14

[72] K. Yancey. *BSD Kernel Developer's Manual: microuptime(9)*, July 2013. 64

[73] S. Zander and G. Armitage. Minimally-intrusive frequent round trip time measurements using synthetic packet-pairs. Technical Report TR 130730A, Centre for Advanced Internet Architectures, 2013. 48, 49

[74] S. Zander and G. Armitage. TEACUP v0.8 - A System for Automated TCP Testbed Experiments. Technical Report 150210A, Centre for Advanced Internet Architectures, February 2015. 48